HAL
open science

Calcul Haute Performance : Caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs

Jean Pourroy

▶ **To cite this version:**

Jean Pourroy. Calcul Haute Performance : Caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs. Théorie de l'information et codage [math.IT]. Université Paris-Saclay, 2021. Français. NNT : 2021UPASM020 . tel-03249275

HAL Id: tel-03249275

https://theses.hal.science/tel-03249275v1

Submitted on 21 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Calcul Haute Performance : caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs

Thèse de doctorat de l'Université Paris-Saclay

Ecole Doctorale de Mathématique Hadamard (EDMH) n° 574
Spécialité de doctorat : Mathématiques appliquées
Unité de recherche : Université Paris-Saclay, CNRS, ENS Paris-Saclay, Centre Borelli, 91190, Gif-sur-Yvette, France.
Référent : Université Paris-Saclay

Thèse présentée et soutenue à Paris-Saclay, le 26 mars 2021, par

Jean POURROY

Composition du jury

Jean-Michel Ghidaglia Professeur des universités, ENS Paris-Saclay	Président
Jean-François Méhaut Professeur (Université Grenoble Alpes), CNRS (LIG)	Rapporteur
Patrick Amestoy Professeur (ENSEEIHT), INP Toulouse (INPT-IRIT)	Rapporteur
Jean-Yves Blanc Architecte informatique en chef, CGG	Examinateur

Direction de la thèse

Christophe Denis Maitre de Conférences, Sorbonne Université (LIP6)	Directeur de thèse
Patrick Demichel Technologue Distingué (DT), HPE	Co-directeur de thèse

Thèse de doctorat

NNT : 2021UPASM020

école normale supérieure paris–saclay

université PARIS-SACLAY

Hewlett Packard Enterprise

CENTRE BORELLI

Fondation mathématique FMJH Jacques Hadamard

Résumé

Titre : Calcul Haute Performance : Caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs

Mots clés : HPC ; Exascale ; Supercalculateur ; Performance ; Optimisation ; Benchmark

Les systèmes d'information et les infrastructures de Calcul Haute Performance (HPC) participent activement à l'amélioration des connaissances scientifiques et à l'évolution de nos sociétés. Le domaine du HPC est en pleine expansion et les utilisateurs ont besoin d'architectures de plus en plus puissantes pour analyser le tsunami de données (simulations numériques, objets connectés), prendre des décisions plus complexes (intelligence artificielle), et plus rapides (voitures connectées, météo).
Dans ce travail de thèse, nous discutons des différents challenges à relever (consommation électrique, coût, complexité) pour l'élaboration des nouvelles générations de supercalculateurs Exascale. Alors que les applications industrielles ne parviennent pas à utiliser plus de 10% des performances théoriques, nous montrons la nécessité de repenser l'architecture des plateformes, en utilisant notamment des architectures énergétiquement optimisées. Nous présentons alors certaines technologies émergentes permettant leur développement : les mémoires 3D (HBM), la Storage Class Memory (SCM) ou les technologies d'interconnexions photoniques. Ces nouvelles technologies associées à un nouveau protocole de communication (Gen-Z) vont permettre d'exécuter de façon optimale les différentes parties d'une application. Cependant, en l'absence de méthode de caractérisation fine de la performance des codes, ces architectures innovantes sont potentiellement condamnées puisque peu d'experts savent les valoriser.
Notre contribution consiste au développement d'une suite de codes (microbenchmarks) et d'outils d'analyse de performance. Les premiers ont pour objectifs de caractériser finement certaines parties de la microarchitecture. Deux microbenchmarks ont ainsi été développés pour caractériser le système mémoire et les unités de calculs. La deuxième famille d'outils permet d'étudier la performance des applications. Un premier outil permet de suivre l'évolution du trafic du bus mémoire, ressource critique des architectures. Un second outil permet d'obtenir le profil des applications en extrayant et caractérisant les boucles critiques (*hot spots*).
Pour profiter de l'hétérogénéité des plateformes, nous proposons une méthodologie en 5 étapes permettant d'identifier et de caractériser ces nouvelles plateformes, de modéliser les performances d'une application, et enfin de porter son code sur l'architecture choisie. Enfin, nous montrons comment les outils permettent d'accompagner les développeurs pour extraire le maximum des performances d'une architecture. En proposant nos outils en « sources ouvertes », nous souhaitons sensibiliser les utilisateurs à cette démarche et développer une communauté autour du travail de caractérisation et d'analyse de performance.

Abstract

Title : High-Performance Computing : Architecture characterization and application optimization for future generations of supercomputers

Keywords : HPC ; Exascale ; Supercomputer ; Performance ; Optimization ; Benchmark

Information systems and High-Performance Computing (HPC) infrastructures play an active role in the improvement of scientific knowledge and the evolution of our societies. The field of HPC is expanding rapidly and users need increasingly powerful architectures to analyze the tsunami of data (numerical simulations, IOT), to make more complex decisions (artificial intelligence), and to make them faster (connected cars, weather). In this thesis work, we discuss several challenges (power consumption, cost, complexity) for the development of new generations of Exascale supercomputers. While industrial applications do not manage to achieve more than 10% of the theoretical performance, we show the need to rethink the architecture of platforms, in particular by using energy-optimized architectures. We then present some of the emerging technologies that will allow their development : 3D memories (HBM), Storage Class Memory (SCM) or photonic interconnection technologies. These new technologies associated with a new communication protocol (Gen-Z) will help to optimally execute the different parts of an application. However, in the absence of a method for fine characterization of code performance, these emerging architectures are potentially condemned since few experts know how to exploit them.

Our contribution consists in the development of benchmarks and performance analysis tools. The first aim is to finely characterize specific parts of the microarchitecture. Two microbenchmarks have thus been developed to characterize the memory system and the floating point unit (FPU). The second family of tools is used to study the performance of applications. A first tool makes it possible to monitor the memory bus traffic, a critical resource of modern architectures. A second tool can be used to profile applications by extracting and characterizing critical loops (*hot spots*).

To take advantage of the heterogeneity of platforms, we propose a 5-step methodology to identify and characterize these new platforms, to model the performance of an application, and finally to port its code to the selected architecture. Finally, we show how the tools can help developers to extract the maximum performance from an architecture. By providing our tools in open source, we want to sensitize users to this approach and develop a community around the work of performance characterization and analysis.

Remerciements

Ces premiers mots de ce manuscrit sont aussi les derniers que j'écris de ce (long ?) manuscrit. Ce sont sûrement les plus importants et me permettent de remercier toutes les personnes qui ont rendu cette aventure possible.

Mes premiers remerciements vont à mon jury qui a accepté d'évaluer mon travail. Je remercie mes deux rapporteurs, **Patrick Amestoy** (INP Toulouse, INPT-IRIT) et **Jean-François Méhaut** (UGA, CNRS), pour avoir pris le temps de lire avec attention mes travaux et pour leurs retours très constructifs. Je remercie aussi **Jean-Michel Ghidaglia** (Professeur des Universités, ENS Paris-Saclay) et **Jean-Yves Blanc** (CGG) d'avoir accepté de faire partie de mon jury.

Je souhaite aussi remercier mes deux directeurs de thèse, **Patrick Demichel** (HPE) et **Christophe Denis** (Sorbonne Université, LIP6) pour leur soutien durant ces quatre années. Merci à **Jean-Jacques Braun** (HPE), **Jean-Luc Assor** (HPE), et **Jean-Michel Ghidaglia** de m'avoir fait confiance depuis le premier jour. Sans votre investissement, les nombreuses difficultés auraient eu raison de cette collaboration entre HPE et l'ENS Paris-Saclay et je souhaite vous remercier sincèrement d'avoir rendu cette thèse possible.

Je remercie aussi les deux administratrices du Centre Borelli (CMLA), **Véronique Almadovar** et **Alina Müller**, ainsi qu'**Agnès Desolneux** (Directrice de recherche CNRS, ENS Paris-Saclay) pour m'avoir épaulé à distance et aidé à relever les nombreuses difficultés administratives pouvant être rencontrées durant un doctorat.

Ce manuscrit aurait été très différent sans l'aide de quatre personnes en particulier que je souhaite remercier. **Patrick Demichel**, merci de m'avoir suivi durant ces quatre années et fait profiter de ton expérience du domaine du HPC, de l'analyse et de l'optimisation de code. J'ai essayé de retranscrire au mieux ce que tu m'as appris dans ces 320 pages, même s'il en avait fallu le double pour décrire seulement l'état de l'art. Tes nombreux conseils, toujours constructifs, m'ont permis d'avancer et m'ont aidé à retrouver la confiance en moi dont j'ai pu manquer à certains moments. J'espère que tu trouveras dans ce manuscrit une part de cette vision de l'Exascale que tu construis depuis toutes ces années. Je suis très fier d'avoir été ton premier thésard, et souhaite toute la réussite possible à ta prochaine *étudiante* qui semble avoir déjà bien commencé.

Je remercie **Thomas Guntz**, *mon copilote, mon partenaire de B.U. et mon duo queue de la rédaction* [Gun20]. Le destin voulait que nous traversions cette épreuve ensemble, après l'**IUT d'Aix-en-Provence**, nous nous sommes retrouvés à l'**ENSIMAG** pour finalement passer des centaines d'heures ensemble à la **Bibliothèque Universitaire Joseph-Fourrier** pour écrire nos manuscrits. Sans ton soutient et celui de la Brasserie du **Martin's Café** (Leboucher x2), je ne pourrais pas dire si j'y serai arrivé, mais on l'a fait (poke Nashor).

Je remercie aussi ma tante, **Geneviève Pourroy** (Directrice de recherche, CNRS) pour

ses nombreuses relectures du manuscrit, et son aide importante (même si tu en doutes) pour l'organisation de ses nombreuses sections, sous-sections, paragraphes...

Aussi, je remercie **Thibault Lambert** pour ses nombreuses corrections orthographiques et grammaticales.

Ces 4 années de thèse chez HPE, et les 3 années d'école d'ingénieur la précédente m'ont fait rencontrer de nombreuses personnes que je souhaite aussi remercier. Je souhaite tout d'abord remercier mes collègues de l'**équipe HPC** de Grenoble, merci pour votre accueil et ces nombreux évènements que nous avons partagés. Ce fut un plaisir de partager ces années avec vous.

Je remercie **Jean-Jacques Braun**, **Pierre Hoffer**, **Nathalie Viollet** et **Philippe Vettori** pour leur encadrement durant ces premières étapes de ma carrière. Merci à **Fred** et **Sébastien**, ce fut un réel plaisir de travailler avec des personnes aussi expérimentées que vous. J'ai beaucoup appris à votre contact et vous remercie pour le temps que vous m'avez consacré et les nombreuses discussions que nous avons partagées. Je remercie **Sorin** pour ton aide et tes conseils tout au long de mon parcours. Merci à **Flo** pour tous ces cafés (mais pas que) partagés au patio. Merci à mon meilleur voisin de bureau, **Xavier**, pour ta bonne humeur et ta joie de vivre. J'espère qu'on aura l'occasion de se croiser sur la route, à vélo. Je souhaite aussi remercier chaleureusement **Hana** pour ces 7 années que nous avons partagés. Tes conseils et ton aide quand j'en avais besoin m'ont été des plus précieux. Je te souhaite toute la réussite que tu mérites dans la suite de ta longue carrière et espère que nos chemins se recroiseront.

Avoir pu travailler dans une entreprise comme HPE a été une première expérience incroyable. L'avantage principal d'une telle entreprise est bien de pouvoir se retrouver le midi au **Foot Loisirs**. Je remercie donc mes partenaires **Max**, **Titi** et **Ben** pour leurs passes (toujours décisives). J'espère qu'on aura l'occasion de rechausser les crampons ensemble ou de partager une bière devant un match de Paris. Je pense aussi à **Paul**, **Pierre-Antoine** et **Max**, qui ont fait de ces années chez HPE un véritable plaisir et sans qui les matchs n'ont plus la même saveur.

Je souhaite aussi remercier de sincères remerciements à tous les enseignants que j'ai pu croiser durant mes études. Je pense notamment à **Andreea Dragut**, **Patricia Gaëtan**, **Marc Laporte**, **Wojciech Bienia**, **Joëlle Thollot** qui m'ont apporté leur soutien à chaque étape de mes études et qui m'ont transmis leur volonté d'enseigner et de partager le plus largement possible la connaissance. Ces expériences m'ont motivé pour enseigner durant trois ans à l'**IUT de Grenoble** auquel j'adresse mes remerciements et notamment à **Jérôme Goulain** et **Philippe Martin**.

La longue aventure que représente la thèse ne serait pas soutenable sans le soutien de ses amis. Merci à mes amis **Flo**, **Manu**, **Tom** et **Yanni** pour tous ces moments qui m'ont permis de m'évader de la thèse durant ces week-ends à lancer quatre boules ou à taper du pied sur des musiques électroniques à rythmes répétitifs. Les soirées, c'est encore mieux quand c'est nous qui

les organisons. Pour ça, je remercie **Shiki**, **Tom**, **Uppah**, **Hugo** et tous autres les bénévoles de **Carton-Pâte Records** pour toutes ces soirées organisées et ces bons moments partagés.

Merci à **Elora**, **Zoé**, **Pierre** et **Cathy** pour votre gentillesse et ces super moments passés à vélo à travers les Alpes ou en ski en Norvège.

Au temps du numérique et des confinements, les amis en ligne sont aussi importants pour s'échapper quelques heures après une longue journée de travail. Je remercie donc toute la **Team RTC** : **Gushu**, **D1aucy**, **Nilure**.

Merci à ma famille. Merci à mes **parents** pour l'éducation qu'ils m'ont donnée et leur soutien tout au long de mes 8 années d'étude. Merci à mon frère **Vincent** et à ma soeur **Clara** de toujours avoir été là pour moi. Je vous aime.

Pour finir, je remercie **Anna** qui m'a soutenu durant ce long parcours. On ne se rend pas compte de la place que peut prendre la thèse dans une vie de couple, mais tu as toujours été là pour m'aider et me changer les idées. Sans toi et ton soutien, je ne pense pas que j'aurai tenu, je te remercie sincèrement pour tout ce que tu m'as apporté.

Table des matières

Glossaire **xi**

Acronymes **xv**

Notations **xvii**

1 Introduction générale **1**
1.1 Supercalculateur exascale 2
1.2 Travail de thèse 6
1.3 Contributions et plan du manuscrit 9

2 Calcul Haute Performance **15**
2.1 Introduction au Calcul Haute Performance (HPC) 16
2.2 Évolution de la performance des supercalculateurs 34
2.3 Opportunités pour le développement de nouveaux supercalculateurs 58
2.4 Caractérisation et analyse de performance des architectures 86
2.5 Conclusion 110

3 Développements logiciels **113**
3.1 Introduction 114
3.2 Benchmark mémoire 118
3.3 Benchmark d'unité arithmétique 143
3.4 Monitoring du bus mémoire 166
3.5 Profil de l'exécution d'instructions 172
3.6 Conclusion 182

4 Méthodologie pour l'analyse de performance et le portage de code **185**
4.1 Introduction 186
4.2 Étape 1 : Rechercher les dernières innovations technologiques 190
4.3 Étape 2 : Caractériser les architectures 195
4.4 Étape 3 : Extraire les noyaux et modéliser leur performance 199
4.5 Étape 4 : Sélectionner la plateforme adaptée 203
4.6 Étape 5 : Porter et optimiser le code 205
4.7 Conclusion et perspectives 212

5 Conclusion générale **215**
5.1 Problématique 216
5.2 Contributions principales de la thèse 217
5.3 Conclusion 220

Annexes 225

Annexe A Architecture des processeurs 225
A.1 Le circuit logique . . . 226
A.1.1 Les transistors . . . 226
A.1.2 Les portes logiques . . . 226
A.1.3 Algèbre de Boole . . . 227
A.1.4 Circuits logiques . . . 227
A.1.5 Les mémoires RAM . . . 229
A.1.6 Évolutions des transistors . . . 230
A.2 L'architecture . . . 232
A.2.1 Architecture de processeur : origine et évolutions majeures . . . 233
A.2.2 Améliorer l'efficacité de l'exécution . . . 237
A.2.3 Accélérer l'exécution des instructions . . . 239
A.2.4 Exécuter les instructions en parallèles . . . 246
A.3 Hiérarchie mémoire . . . 254
A.3.1 Mémoire principale . . . 255
A.3.2 La hiérarchie mémoire . . . 258
A.3.3 Caches . . . 264
A.4 Mémoire virtuelle . . . 272
A.4.1 La pagination . . . 274
A.4.2 Memory Management Unit (MMU) . . . 277

Annexe B Les compteurs matériels 283
B.1 Les compteurs . . . 283
B.1.1 Les compteurs matériels . . . 283
B.1.2 Mode de mesure : comptage et échantillonnage . . . 285
B.1.3 Compteurs matériels des architectures Intel . . . 286
B.2 Unité de suivi de performance Intel : les PMU . . . 289
B.2.1 PMU core . . . 290
B.2.2 PMU uncore . . . 292
B.3 Discussion . . . 297
B.4 Conclusion . . . 299

Glossaire

Benchmark Code, ou un ensemble de codes, permettant de mesurer la performance d'une solution et d'en vérifier ses fonctionnalités. 7, 87, 88, 111, 114, 117, 143, 187, 196, 218, 219, 222

Compteur Matériel Les compteurs matériels de performance (ou *hardware counters*) désignent les registres matériels du processeur utilisé pour enregistrés et compter les évènements matériels et logiciels arrivant sur la microarchitecture (voir PMU). 283, 296

Compute Bound Le terme "limité par le calcul" (*compute bound*) fait référence à une application (ou fonction) dont le temps d'exécution est principalement déterminé par la performance de calcul du processeur. Lorsqu'un code n'est pas *compute bound* il est généralement memory bound. xii, 107, 124, 125, 145, 189, 199, 200

Exascale désigne la nouvelle génération de plateforme capable d'exécuter un 10^{18} FLOPS (un exaFLOPS) (10^{18} opérations à virgule flottante par seconde) 2, 3, 45, 58, 60, 73, 77, 78, 86, 94, 186, 220

FMA instruction processeur réalisant une addition et une multiplication entre trois valeurs a, b et c tel que $a = a + b * c$. L'unité matérielle responsable de l'exécution d'une telle instruction s'appelle un multiplicateur-accumulateur (MAC) xv

FPU Une unité de calcul à virgule flottante Floating Point Unit) est une partie d'un processeur, spécialement conçue pour effectuer des opérations sur des nombres à virgule flottante. xv

Framework infrastructure logicielle désignant un ensemble de composants logiciels établissant les fondations d'un logiciel. 92

Goulot D'étranglement 105, 200, 257

Hot Spot Désigne une région d'un programme où une grande proportion d'instructions sont exécutées pendant l'exécution d'une application. 11, 87, 95, 107, 117, 172, 173, 176, 189, 190, 199, 203, 219

HPC Le but du HPC est de paralléliser des applications scientifiques à destination de ressources informatiques telles que les supercalculateurs xv

Kernel Un noyau de calcul (*kernel*) est une partie de code restreinte d'une application, qui remplit une fonction clairement définie. En Calcul Haute Performance (HPC), les *kernels* sont des zones de codes de calculs intensifs responsables de la majorité du temps d'exécution d'une application. 8, 110, 115, 145, 147, 156, 164, 167, 186, 189, 191, 199–202, 205, 207

Memory Bound Le terme "limité par la mémoire" (*memory bound*) fait référence à une application (ou fonction) dont le temps d'exécution est principalement déterminé par la performance du système mémoire. Lorsqu'un code n'est pas *memory bound* il est généralement compute bound. xi, 107, 124, 128, 189, 199, 200

Memory Gap Traduit l'écart de performance entre la performance des processeurs et celle du système mémoire. 60

Miss Un défaut de cache (ou *miss*) est un événement se produisant lorsqu'une donnée à accéder n'est pas présente dans la mémoire cache du processeur. Celle-ci doit alors être chargée depuis la mémoire principale. 8, 11, 116, 128, 139, 166, 168, 169, 218, 292

MPI Standard de communication pour des programmes parallèles sur des systèmes à mémoire distribuée xv

PMU La PMU est un matériel du processeur responsable de mesurer la performance de celui-ci à l'aide de compteurs matériels spécialisés (*hardware counters*). xv

Prélecteur Mémoire Dispositif matériel du processeur permettant d'anticiper les accès mémoire en chargeant les données (ou les instructions) depuis la mémoire vers le processeur (en mémoire cache) avec qu'elles ne soient réellement utilisées. Ce mécanisme, aussi appelé *memory prefetcher*, permet de réduire la latence d'accès. 11, 118, 119, 122, 123, 125, 128, 131–134, 218

Stride Certains algorithmes réalisent des accès mémoires en accédant aux données par des sauts, de taille régulière, en entre deux adresses mémoire. Ces sauts sont appelés des strides. 8, 93, 118–120, 125, 126, 132, 141, 187, 217, 218

Thread ou processus léger ou tâche est similaire à un processus. Les threads d'un même processus se partagent le même espace mémoire. 31, 99

Translation Lookaside Buffer ou répertoire de pages actives (TLB) permet d'accélerer la traduction des adresses virtuelles. 7, 120, 126, 139

Acronymes

ALU *Arithmetic Logic Unit* ou unité arithmétique et logique 144

ASIC *Application-Specific Integrated Circuit* ou circuit intégré propre à une application 24

DSP *Digital Signal Processor* ou processeur de signal numérique 24

ExaFLOPS 10^{18} FLOPS xi, 3, 45

FLOP *Floating Point Operation* ou opération à virgule flottante 8, 20, 42, 43, 86, 143, 176, 192, 195, 204, 217, 239, 255, 284, 288, 289

FLOPS *Floating Point Operation per Second* ou nombre d'opérations à virgule flottante par seconde (FLOP/s) 2, 4, 34, 35, 40, 42, 46, 50, 75, 89, 145, 156, 157, 174, 192, 193, 199

FMA *Fused Multiply-Add* ou multiplication-addition fusionnées (*voir glossaire*:FMA) 27, 109, 147, 157, 162, 192, 194, 195, 197, 206

FPGA *Field Programmable Gate Arrays* ou réseaux logiques programmables 23, 56

FPU *Floating Point Unit* ou unité de calcul à virgule flottante (*voir glossaire*:FPU) 8, 41, 143–145, 162, 163, 165, 192, 210, 218, 219, 228, 244

GPU *Graphics Processing Unit* ou processeur graphique 22, 23, 186, 190

HPC *High Performance Computing* ou Calcul Haute Performance (*voir glossaire*:HPC) xii, 1, 2, 7, 8, 13, 15, 18, 21, 86, 96, 111, 191, 215, 236

IPC Instructions Par Cycle 11, 128, 136, 137, 151, 154–157, 160, 179, 208, 210, 287

MPI *Message Passing Interface* ou interface de passage de message (*voir glossaire*:MPI) 30, 210

PMU *Performance Monitoring Unit* ou unité de suivi de performance (*voir glossaire*:PMU) 167, 169, 286

Notations

$\texttt{EQUILIBRE}_{\texttt{architecture}}$	L'équilibre arithmétique d'une architecture 204
$\texttt{FLOP}_{\texttt{cycle}}$	nombre d'opérations à virgule flottante (FLOP) exécuté par cycle 192, 193
$\texttt{FLOPS}_{\texttt{max}}$	performance de calcul maximale mesurée d'une unité de calcul, mesurée en FLOPS 145, 157, 162, 163, 165, 195–198, 202
$\texttt{FLOPS}_{\texttt{peak}}$	performance de calcul théorique d'une unité de calcul, mesurée en FLOPS 107, 108, 156, 157, 165, 192, 193, 195, 196, 200
$\texttt{MEMORY}_{\texttt{max}}$	débit mémoire maximale mesuré du bus mémoire, mesuré en GB/s 196, 197, 201, 202, 211
$\texttt{MEMORY}_{\texttt{peak}}$	débit mémoire théorique du bus mémoire, mesuré en GB/s 107, 108, 192, 196, 200, 201
$\texttt{OI}_{\texttt{kernel}}$	l'intensité opérationnelle d'un kernel correspond au ratio entre le nombre d'opérations exécutées et la quantité de données transférées nécessaire. 199, 200, 202, 204
$\texttt{TEMPS}_{\texttt{optimal}}$	Temps optimal calculé pour exécuter un kernel 201, 203, 205–207, 211, 213

Chapitre 1

Introduction générale

Le Calcul Haute Performance (HPC) est le domaine consistant à regrouper des milliers de ressources informatiques (processeurs, mémoire, stockage, réseau) pour exécuter des applications de simulations numériques complexes ou permettre le traitement massif de données. Ces simulations contribuent au développement de nouveaux produits (voitures, avions), à l'extraction d'énergies (recherche pétrolière, conception d'éolienne). Le HPC est un outil inhabituel, car il ne se limite pas à un domaine particulier et possède des applications dans de nombreux domaines :

— En physique théorique, les expérimentations réalisées au Grand Collisionneur de Hadrons (LHC) demandent de grandes puissances de calcul pour analyser les données générées chaque seconde par 600 millions de collisions. L'analyse de ces données en un temps raisonnable est rendue possible grâce à une plateforme distribuée entre plusieurs pays formés d'un million de coeurs et d'un exabyte de stockage (10^{18} octets). Cette plateforme de calculs a permis d'importantes découvertes comme celle du boson de Higgs [Bel+17].

— Dans le domaine de la santé, le Living Heart Project [Bai+14] a pour objectif de modéliser un coeur pour aider à comprendre son fonctionnement. Depuis 2017, les fabricants peuvent utiliser cette modélisation pour tester un nouveau traitement avant de l'utiliser sur un patient. Un second projet nommé *Human Brain Project* [Mar12] a pour objectif de simuler d'ici 2024 le fonctionnement d'un cerveau humain permettant de développer des traitements pour les maladies neurologiques.

— En mécanique des fluides, les simulations numériques sont utilisées dans l'aéronautique (maximiser le portage des ailes), la recherche d'hydrocarbures (modélisation 3D des fonds marins), la météorologie...

Ainsi, le HPC participe activement à l'amélioration des connaissances scientifiques, mais aussi à l'évolution de nos sociétés, par exemple, en rendant nos vies plus sécurisées en anticipant certaines catastrophes naturelles (météorologie, épidémiologie, développement de médicaments [1], etc.). Ces simulations nécessitent d'énormes quantités de calculs pour être exécutées et utilisent des plateformes informatiques constituées de milliers de ressources de calculs appelées supercalculateurs. L'amélioration de la performance de ces plateformes année après année a permis de réaliser de nombreuses avancés dans les domaines cités ci-dessus. En effet, leur performance a évolué exponentiellement ces 30 dernières années. Une montre connectée récente possède une puissance de calcul supérieure à la plateforme la plus puissante développée en 1985 (Cray-2),

1. En mars 2020, l'Institut Pasteur et les équipes de recherche de Sorbonne Université ont utilisé le supercalculateur Jean Zay, conçu par HPE, pour étudier la structure moléculaire du SARS-CoV-2 afin d'aider au développement de médicaments pour soigner le coronavirus - http://www.genci.fr/fr/node/1043.

alors utilisée dans le domaine nucléaire.

Le développement de plateformes de calculs plus puissantes va aider notre société à relever les nombreux défis qui lui font face : changement climatique (anticiper les catastrophes naturelles, étude du réchauffement des océans), recherche médicale (médecine personnalisée, recherche biomoléculaire), villes intelligentes, etc. Cette nouvelle génération de supercalculateurs permettra d'obtenir des réponses plus fiables et plus rapidement et permettra notamment d'analyser les grands volumes de données générées par les objets connectés. Ces analyses peuvent utiliser des algorithmes d'apprentissage automatique qui ne deviennent efficaces qu'après avoir suivi de nombreux entraînements, ces derniers nécessitant de grandes puissances de calcul. Ainsi, si la société veut continuer à profiter de nouvelles améliorations, la performance des supercalculateurs doit continuer à évoluer.

1.1 Supercalculateur exascale

Actuellement, le défi de l'industrie des semi-conducteurs est la construction d'une plateforme capable d'exécuter 10^{18} opérations par seconde nommée supercalculateur exascale. L'accès à une telle plateforme permettra alors d'exécuter les applications actuelles sept fois plus rapidement [2] (ou nécessitant sept fois plus d'opérations) que les systèmes *petascale* les plus puissants d'aujourd'hui. L'obtention du premier supercalculateur exascale est très convoitée et représente un symbole de réussite pour son acquéreur (pays, université, industrie), ainsi que pour l'entreprise qui aura réalisé sa construction. L'industrie du HPC, mais aussi les nations, se sont donc lancées dans cette course effrénée en y allouant des moyens considérables. Nous pouvons notamment citer les investissements réalisés par les États-Unis (PathForward, 400 millions d'euros) ou l'Union européenne (EuroHPC, 486 millions d'euros). Ces projets sont réalisés en partenariat avec des entreprises privées du domaine HPC (dont HPE) qui financent d'un même montant chaque projet.

1.1.1 Les défis d'Exascale

Pour comprendre la complexité des défis à relever pour construire une plateforme exascale, le classement du Top500 [3] peut être consulté. Le Top500 classe deux fois par an les 500 supercalculateurs les plus puissants de la planète à l'aide d'une application de calcul intensif High-Performance Linpack (`HPL` [DLP03]). Cette application permet d'évaluer la puissance de calcul d'une plateforme, mesurée en nombre d'opérations à virgule flottante par seconde (FLOPS). Ce classement existe depuis 1993 et permet de mesurer et d'apprécier l'évolution de la performance des supercalculateurs les plus puissants tous les 6 mois.

La figure 1.1 présente l'évolution de la performance cumulée des 500 supercalculateurs apparaissant dans chaque classement du Top500. De 1993 à 2012, cette performance cumulée a

2. En novembre 2019, le supercalculateur le plus puissant de la planète est capable d'exécuter 148×10^{15} opérations par seconde.

3. Classement du Top500 - `https://www.top500.org/`

évolué d'un facteur 1000 tous les 11 ans pendant près de 20 ans. Durant cette période, les processeurs ont pu profiter de nombreuses évolutions technologiques (voir figure 1.2) : augmentation du nombre de transistors et de la fréquence des processeurs, augmentation du nombre de coeurs ou encore l'implémentation d'une hiérarchie mémoire accélérant les accès aux données.

Pour atteindre une puissance de 10^{18} FLOPS (un exaFLOPS), nous pouvons ainsi nous demander s'il ne suffirait pas d'attendre les prochaines générations de processeurs et d'augmenter le nombre de serveurs formant un supercalculateur. Malheureusement, nous constatons, depuis 2012, un ralentissement de l'évolution des performances des supercalculateurs. Il faut depuis cette date attendre 23 ans, contre 11 jusqu'en 2012, pour voir la performance des supercalculateurs évoluer du même facteur 1000. Ce ralentissement est dû à un ensemble de contraintes qu'il n'est plus possible d'éviter, dont les deux principales sont :

— la fin de la validité de loi de Moore [Moo75], qui prévoyait l'augmentation du nombre de transistors chaque année ;
— la fin de la loi de Dennard [Den+74], qui assurait l'augmentation de la fréquence de chaque nouvelle génération de processeur.

De plus, les systèmes mémoires et les capacités de traitement des processeurs ont évolué inégalement. Lorsque la performance calculatoire augmentait de 50% chaque année, le débit mémoire n'augmentait que de 23%. Ce déséquilibre entre ces deux parties fondamentales de l'architecture Von Neumann a donné naissance au "mur de la mémoire" [Wil01]. À cause de cette inégalité d'évolution, les supercalculateurs modernes ne parviennent à extraire qu'une fraction des performances théoriques des architectures. Celle-ci est même couramment inférieure à 10% [Oli+05] lors de l'exécution d'applications HPC utilisées dans le domaine du calcul scientifique.

En plus des freins présentés ci-dessus, de nombreux défis supplémentaires doivent être relevés afin de poursuivre le rythme d'évolution des performances des supercalculateurs. Plusieurs études [Sut05 ; Kog+08 ; Luc+14 ; HPE16] prédisent depuis plusieurs années les principales difficultés rencontrées pour développer une plateforme exascale :

— amélioration de l'efficacité énergétique : la consommation électrique des plateformes doit être réduite de plusieurs facteurs pour permettre l'exécution de 10^{18} opérations par secondes. Le défi énergétique s'applique à l'ensemble de la plateforme (processeur, mémoire, systèmes d'interconnexion et de refroidissement) ;
— développement de nouvelles technologies mémoire : de nouvelles mémoires doivent être développées permettant de fournir un débit suffisamment élevé et des capacités adaptés à l'exécution d'applications *exascale* ;
— amélioration de la performance (débit, latence) des technologies d'interconnexion ;
— résistance aux erreurs, sécurité des données sensibles, productivité des utilisateurs, etc.

Le principal défi est celui de l'énergie. Avec les technologies actuelles, un supercalculateur exascale consommerait plusieurs centaines de mégawatts, équivalent à la consommation électrique de deux villes comme Gap (19 000 foyers). Pour des raisons de coût et de faisabilité, l'objectif de l'industrie HPC est de construire une plateforme exascale consommant entre 20 et 30 MW.

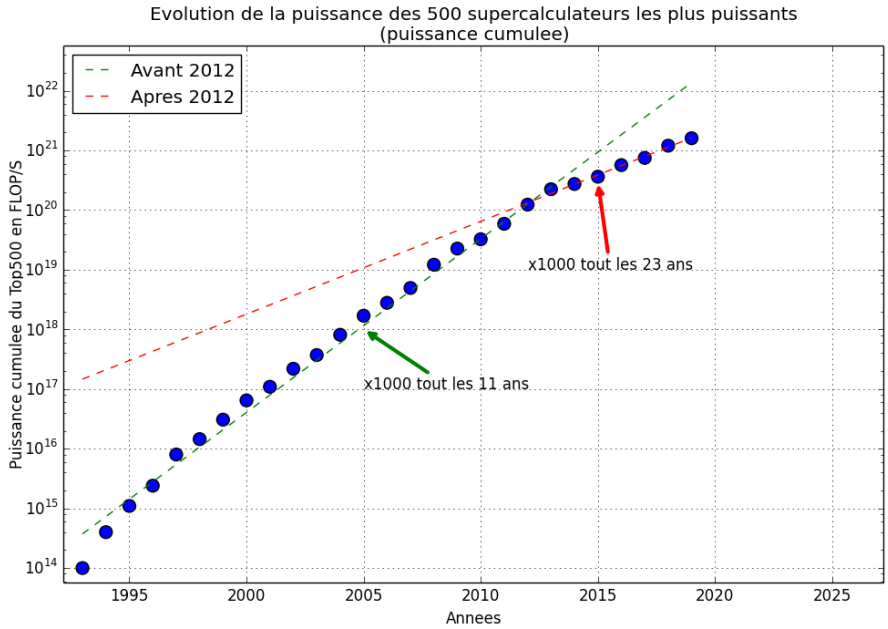


FIGURE 1.1 – Évolution de la performance cumulée des 500 supercalculateurs les plus puissants au monde, mesurée en FLOPS à l'aide de l'application `HPL` [DLP03].

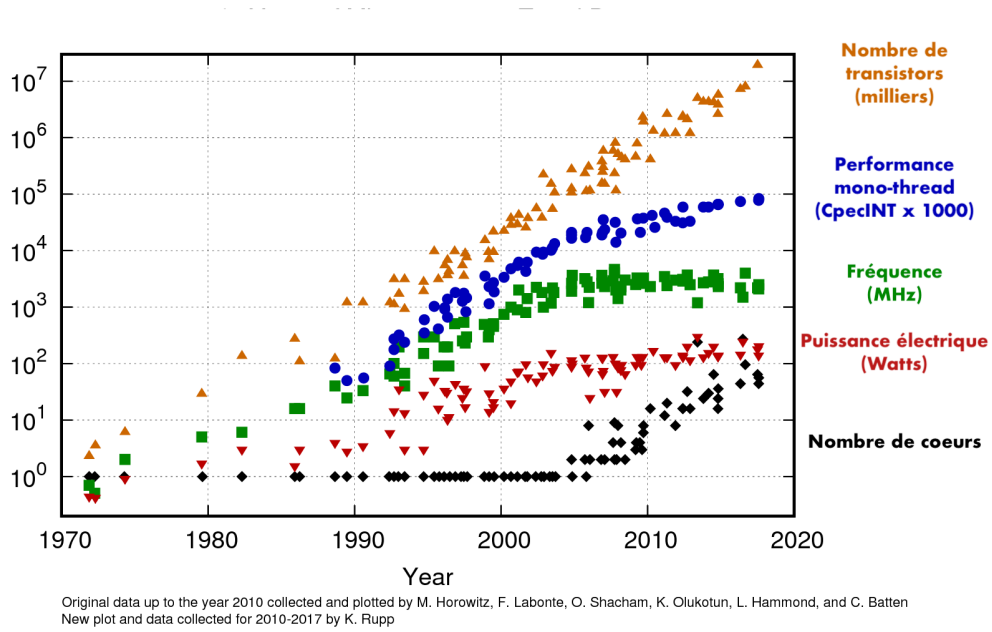


FIGURE 1.2 – Évolution des principales caractéristiques des processeurs (données originales tirées de [Rup15] [4]).

4. Les données sont accessibles sur le dépôt GitHub de l'auteur : https://github.com/karlrupp/microprocessor-trend-data

Le challenge est donc de construire une plateforme sept fois plus puissante que le supercalculateur le plus puissant actuel, tout en étant trois fois plus efficace énergétiquement. Les déplacements de données à l'intérieur (entre la mémoire et le processeur) et à l'extérieur (entre deux serveurs) des serveurs sont les principales opérations consommatrices d'énergie. Cette consommation peut être en partie expliquée par la complexité des microarchitectures, mais aussi par les technologies d'interconnexion utilisées. La complexité des processeurs récents, due aux nombreuses améliorations qu'ils ont connues ces dernières années (hiérarchies mémoire, *pipeline*, unité de préchargement mémoire), est à l'origine de la difficulté des applications à extraire plus d'une fraction des performances théoriques disponibles.

1.1.2 Les opportunités

L'incapacité des technologies actuelles à réaliser efficacement des calculs et l'explosion du nombre de données à traiter nécessite de repenser les matériels utilisés et l'architecture des systèmes informatiques. Heureusement de nouvelles technologies sont développées et représentent de réelles opportunités pour le développement des prochaines générations de plateformes de calculs.

La première opportunité vient du développement de nouvelles technologies mémoire permettant de combler la différence d'évolution de performance constatée entre les processeurs et les mémoires. La faiblesse des débits mémoires et la consommation électrique du système mémoire obligent les architectures à évoluer drastiquement :

— la première solution est de rapprocher ces espaces mémoire des zones de calculs en plaçant la mémoire sur les puces de calculs (On Package Memory) ou en réalisant les calculs directement en mémoire (In Memory Computing) ;

— la seconde solution est de développer de nouvelles technologies mémoire afin de combler le trou séparant les caractéristiques des mémoires DRAM et des disques de stockage flash. Pour cela des technologies NVRAM (non-volatile RAM) sont développées en s'appuyant notamment sur de nouvelles technologies mémoire SCM (Storage Class Memory).

Une deuxième opportunité technologique pouvant aider à l'élaboration d'une plateforme *exascale* est le développement de technologies photoniques. Ces technologies utilisent la lumière pour transférer des informations et permettre d'atteindre des débits élevés avec des latences faibles. Une autre qualité de ces technologies est d'être indépendantes de la distance. Bien que la génération d'un signal optique ou électronique nécessite la même énergie, celle-ci n'évolue que faiblement avec la distance de communication. Le coût énergétique d'un accès mémoire local est alors proche de celui d'un accès distant.

Ces nouvelles technologies, associées aux méthodes de co-design qui associent les utilisateurs et les fabricants, vont permettre de développer de nouvelles architectures optimisées pour certains algorithmes. L'utilisation d'architectures spécialisées pour l'exécution de certains codes dans une même plateforme a été démocratisée par l'utilisation massive des GPU. Cette approche de plateformes *hétérogènes* doit être poursuivie avec de nouvelles architectures très différentes de celles actuellement utilisées : les accélérateurs produits par Google (TPU), NEC

(SX-Aurora), ou encore Pezy (PEZY-SC2).

Afin de faciliter l'intégration de ces nouvelles technologies (mémoires, processeurs) provenant de différents constructeurs, un nouveau protocole est en cours de développement. Nommé Gen-Z, ce projet est développé par 70 des plus grandes sociétés des technologies de l'information (IT). Il repose sur l'utilisation d'un protocole de communication à sémantique mémoire `load/store`. Gen-Z permettra d'adresser un espace mémoire mille fois plus grand que notre espace numérique actuel et interconnecter 16 millions d'objets sur une même matrice de commutation (*fabric*). Les premières versions d'architecture Gen-Z permettront d'atteindre des débits mémoires de plusieurs téraoctets par seconde contre quelques centaines de gigaoctets actuellement.

1.2 Travail de thèse

De nouvelles technologies vont permettre à l'industrie d'implémenter cette vision du calcul extrême qui consiste à l'exécution d'applications demandant des ressources dépassant de très loin les moyens disponibles actuels (villes intelligentes, analyse de données, etc.). Afin que l'exécution de ces applications puisse être réalisée dans des contraintes raisonnables (temps, budget, complexité), le calcul extrême nécessite de revoir l'architecture des plateformes de calculs de manière holistique. Face à l'explosion de la quantité de données générées, il est indispensable de repenser l'architecture des centres de données, mais aussi de la plateforme permettant leur collection, leur transfert et leur analyse.

1.2.1 Problématique

Pour subvenir au besoin de puissance grandissant et ne pouvant compter sur les anciens leviers à sa disposition, l'industrie doit s'appuyer sur l'utilisation de nouvelles technologies émergentes qui permettront de faire des sauts de performance de plusieurs facteurs (10, 100 voir 1000). Grâce au protocole universel Gen-Z et aux nouvelles technologies présentées précédemment, de nombreuses architectures vont être développées et seront très différentes de celles que nous utilisons actuellement (processeurs x86, GPU). Ces processeurs seront alors très efficaces pour l'exécution de certains types d'opérations et d'algorithmes. Afin de répondre aux besoins de performances supplémentaires et relever le défi de l'efficacité énergétique, il est nécessaire d'utiliser les architectures les plus adaptées pour l'exécution d'une application. Pour ce faire, trois tâches principales doivent être réalisées :

1. La première tâche est de caractériser l'ensemble de ces architectures pour trouver leurs forces et leurs faiblesses : débit et latence mémoire, capacité calculatoire, performance du système mémoire pour certains motifs d'accès, etc. Ces nouvelles architectures pouvant être très différentes de celles que nous utilisons actuellement, il est indispensable de disposer d'un ensemble d'outils permettant de les caractériser pour différents types d'algorithmes.
2. La deuxième tâche consiste à modéliser les performances d'une application pour en connaître ses besoins (débit et/ou latence mémoire, calcul vectoriel, système d'intercon-

nexion, stockage, etc.). Les gains de performance ne viendront pas seulement de l'utilisation d'accélérateurs puissants, mais de leur diversité et de la capacité des programmeurs à bien les utiliser. Pour une même application, plusieurs zones de code prenant une part importante dans son exécution peuvent avoir des besoins différents. Il est ainsi nécessaire de pouvoir identifier ces zones et de les modéliser indépendamment.

3. La troisième tâche est de sélectionner les architectures adéquates pour une application et adapter le code pour permettre son exécution. Il est ensuite nécessaire de pouvoir mesurer les performances de l'application et d'appliquer les optimisations adaptées pour obtenir une part significative de la performance théorique.

1.2.2 Travaux existants

Afin de répondre aux tâches présentées ci-dessus, les programmeurs doivent alors avoir à leur disposition un ensemble d'outils permettant de caractériser le comportement des architectures et d'étudier la performance de leurs applications lors de l'exécution.

1.2.2.1 Caractérisation des architectures

Afin de caractériser les architectures et d'en mesurer les performances atteignables, il est courant d'utiliser des benchmarks. En informatique, un benchmark est un code, ou un ensemble de codes, permettant de mesurer la performance d'une solution et d'en vérifier ses fonctionnalités. Dans ce travail de thèse, nous nous intéressons à la caractérisation de deux composants fondamentaux des processeurs utilisés pour l'exécution d'applications HPC : la hiérarchie mémoire et les unités de calcul vectorielles.

Caractérisation du système mémoire : le système mémoire étant la principale ressource limitant la performance des applications HPC, de nombreux travaux ont été menés pour aider à sa caractérisation. Le plus utilisé d'entre eux est le benchmark `STREAM` [McC95]. Ce code utilise 4 noyaux de calculs différents pour mesurer le débit mémoire atteignable par une architecture. Pour cela, un tableau de nombres aléatoires stocké en mémoire est accédé par des instructions de lecture/écriture pour réaliser quatre opérations différentes (copy, scale, add, triad). La simplicité de ces codes permet souvent d'atteindre les débits maximaux du bus mémoire et fait de `STREAM` un benchmark de référence. Un autre benchmark largement utilisé dans l'industrie est celui de `Lmbench` [Sta05]. `Lmbench` est une suite de microbenchmarks utilisée pour mesurer la latence et le débit atteignable par un coeur pour différentes opérations : accès mémoire, ouverture de fichier, création de *pipes*, etc. Il permet aussi de trouver certaines caractéristiques de la microarchitecture comme la taille d'une ligne de cache, la fréquence du processeur ou encore la taille et la latence du répertoire de pages actives (TLB).

Que ce soient `STREAM`, `lmbench` ou d'autres benchmarks mémoires [YPS05 ; Lus+06 ; Gon+10 ; DH13 ; BLK18], aucun code disponible ne permet de caractériser la hiérarchie mémoire lors d'accès mémoire par sauts d'adresses. De nombreuses applications utilisent des accès mémoires

réalisant des sauts d'adresses (stride) de taille constante. Par exemple, le parcours d'un tableau d'objet pour accéder à un certain champ ou le parcours d'une matrice (en ligne ou en colonne). Pour accélérer ces accès, l'unité de préchargement mémoire peut anticiper les prochaines adresses accédées en comprenant le motif d'accès utilisé et commencer leur chargement avant qu'elles ne soient accédées. Pour des applications utilisant ce type d'accès par saut, il est primordial que ce composant fonctionne correctement pour réduire au maximum le défaut de cache (miss) d'une donnée dans le cache lors de son accès. Le benchmark s'approchant le plus de cet objectif est celui de Saavedra [SS95]. Il utilise une taille de saut fixée au début de l'exécution pour accéder à un jeu de données. La taille de ce dernier doit être un multiple d'une puissance de 2 et ne permet pas de dépasser la taille du dernier niveau de cache. Le problème d'une telle approche est de vouloir mesurer tous les niveaux de la hiérarchie simultanément. Les mesures peuvent alors être influencées par différents paramètres de différents niveaux de caches.

Caractérisation des unités de calculs vectorielles : la majorité des applications utilisées en HPC exécutent des opérations à virgule flottante (FLOP). Ces instructions sont exécutées par un matériel spécialisé nommé unité de calcul à virgule flottante (FPU). La capacité des FPU à exécuter plusieurs instructions vectorielles par cycle est alors primordiale pour la bonne performance des applications. La littérature dénombre plusieurs benchmarks couramment utilisés pour mesurer la performance de calcul des processeurs. Le benchmark `HPL` [DLP03] a été développé pour caractériser la microarchitecture d'un processeur lors de la résolution de problèmes algébriques. Ce code permet aujourd'hui de mesurer et classer les supercalculateurs dans le classement du Top500. Cependant, la simplicité du benchmark `HPL` n'est pas représentative d'applications réelles. Ainsi, un ensemble de codes a été rassemblé pour améliorer la caractérisation des supercalculateurs. Nommée `HPCG` [DH13], cette suite de codes permet de couvrir plusieurs motifs de communication et de calculs. Les versions récentes du Top500 publient pour certains supercalculateurs la performance obtenue sur le benchmark `HPCG`. Nous pouvons aussi citer le benchmark `HPCC` [Lus+06] qui est une suite de 6 benchmarks contenant notamment `HPL` et `STREAM`. Un seul exécutable est généré pour l'ensemble de la suite permettant d'éviter l'utilisation de certaines optimisations pour un code particulier (pages larges, réglage BIOS, utilisation d'une fréquence définie, etc.).

Les benchmarks existants se basent sur des noyaux de calcul (kernels) qui permettent de mesurer la performance que d'un ensemble d'opérations restreint. Une grande majorité des benchmarks n'utilisent pas de code assembleur. La performance mesurée peut alors fortement varier avec la qualité du compilateur utilisé. Ainsi, nous constatons le manque d'un outil permettant de valider précisément les performances des FPU. Un tel outil permettrait de mesurer le nombre d'instructions par cycle que le processeur est capable d'exécuter en faisant varier différents paramètres : type d'opérations, taille des instructions vectorielles, mélange d'instructions différentes, ajout de chaînes de dépendances, etc.

1.2.2.2 Suivi de performance des applications

Le deuxième aspect de nos recherches concerne le suivi de performances des applications. L'objectif de celui-ci est de récolter des informations concernant l'exécution de l'application. Ces informations peuvent être obtenues en instrumentant le code (manuellement ou grâce au compilateur) ou en récupérant certaines informations de l'architecture. Pour cela, les processeurs possèdent des registres (compteurs) qui permettent de compter des événements (matériels ou logiciels) avec un faible impact sur l'application exécutée. Chaque famille de processeurs possède un jeu différent de compteurs matériels pouvant avoir des noms différents, rendant leur programmation difficile. Afin de faciliter leur programmation, des interfaces comme `PAPI` [Bro+00] et `Perf Events` [Wea13] ont été développées. Celles-ci permettent de développer des outils sans se préoccuper de l'implémentation matérielle des compteurs facilitant ainsi la portabilité des outils. Nous pouvons notamment citer les outils d'Intel (`VTune`), de Linux (`perf`) ou encore ceux développés au Barcelona Supercomputing Center (`Extrae` [Rod], `Paraver` [Pil+95] et `Dymemas` [LGC97]).

Afin de répondre aux différentes tâches de la problématique traitée dans ce travail de thèse, deux outils principaux sont nécessaires. Le premier doit permettre de suivre l'activité du bus mémoire. Ce bus est une ressource critique des architectures modernes, rendant sa bonne utilisation indispensable. Il est donc nécessaire de posséder un outil permettant de suivre l'état du trafic sur ce bus. Pour cela, des outils existent, mais sont soit propriétaires, soit trop complexes pour être facilement installés sur des architectures novatrices. Un second outil doit être capable d'extraire les zones de codes les plus intéressantes à porter sur ces nouvelles architectures. Les supercalculateurs seront hétérogènes et utiliseront plusieurs types d'accélérateurs adaptés à certaines fonctions. Une même application pourra alors faire appel à plusieurs d'entre eux. Il est donc nécessaire de posséder un outil permettant de caractériser les différentes fonctions d'une même application en extrayant ces zones de codes et en les caractérisant. Ce même outil pourrait alors aussi être utilisé pour comprendre les performances de l'application et appliquer les optimisations adéquates.

1.3 Contributions et plan du manuscrit

L'industrie va faire face à une multitude de révolutions technologiques (nouvelles mémoires, interconnexion photonique, protocole Gen-Z) et les utilisateurs de supercalculateur doivent se préparer à ces changements profonds pour pouvoir les exploiter au maximum. Contrairement aux évolutions précédentes où il suffisait d'installer un nouveau processeur ou d'ajouter des barrettes mémoires, il faut repenser entièrement notre façon d'appréhender les architectures des supercalculateurs et adapter les algorithmes qui y seront exécutés. Cependant, en l'absence de méthode de caractérisation fine de la performance des codes, ces architectures innovantes sont potentiellement condamnées puisque peu d'experts savent les valoriser. En effet, la loi de Moore [Moo75] a permis d'assurer une évolution constante de la performance des processeurs, laissant les domaines de l'analyse et de l'optimisation des codes, alors économiquement non rentables, en

second plan. Ainsi, nous avons constaté à travers de nombreuses rencontres avec les utilisateurs, mais aussi par les expériences internes à HPE, le manque d'outils et de connaissances nécessaires à la réalisation de ce travail.

1.3.1 Contributions

Ce travail de thèse contribue à l'étude du domaine du calcul haute performance. Le déroulement de cette thèse dans un cadre industriel a permis de réaliser un large état de l'art de ce domaine complexe. Nous regroupons et expliquons les principaux défis auxquels les constructeurs font face dans le développement des prochaines générations de supercalculateurs. Afin d'y répondre, nous présentons les principales opportunités technologiques qui vont permettre de relever les défis présentés et notamment de la nécessité d'utiliser des architectures novatrices pour soutenir la contrainte énergétique.

Afin de pouvoir profiter de ces nouvelles technologies, nous proposons dans ce travail une méthodologie en 5 étapes permettant de modéliser les performances d'une application, de les projeter sur de nouvelles architectures et d'en extraire la performance maximale. Pour chaque étape, nous avons développé et sélectionné des outils permettant de répondre aux challenges présentés et qui respectent les critères suivants :

— Les sources de l'outil doivent être disponibles pour permettre de les adapter à des architectures différentes et pour développer une communauté d'utilisateurs autour de ces sujets.
— Les outils doivent être simples, ne cherchant pas à répondre à une multitude de questions, ce qui rend généralement difficile leur portabilité, mais aussi les conclusions pouvant être tirées des résultats obtenus. La simplicité des outils permet aussi de réduire leur dépendance à des bibliothèques externes, souvent difficiles à installer dans les environnements HPC.
— Les outils doivent permettre d'analyser des applications HPC dont l'exécution peut durer plusieurs heures. Il faut aussi réduire au maximum la nécessité de posséder des droits spéciaux (root) pour les utiliser, ce dernier étant rarement disponible pour les utilisateurs de supercalculateurs.

Contrairement à une majorité d'outils existants, ceux développés durant ce travail de thèse n'ont pas vocation d'automatiser entièrement le travail d'analyse. La puissance de ces outils vient de leur utilisation complémentaire. Leur utilisation assume que l'utilisateur possède de solides connaissances. En fournissant une méthodologie permettant de bien les utiliser et en réduisant la complexité de l'outillage, nous espérons que leur adoption auprès des programmeurs sera plus grande. Un outil a été développé pour chaque étape de la méthodologie lorsqu'aucun programme existant ne répondait aux besoins, ou qu'il ne respectait pas les critères énoncés ci-dessus. La principale difficulté concernant l'utilisation ou le développement d'outils concerne leur compatibilité avec différentes architectures. Celles-ci pourront être produites par différents constructeurs (Intel, ARM, IBM, Nvidia, etc.), il était donc primordial d'utiliser ou de développer des outils ayant le plus de chances d'être compatibles avec ces nouvelles architectures.

Ce besoin de compatibilité nous a obligés à restreindre la dépendance des outils à certaines fonctionnalités des processeurs, notamment l'utilisation des compteurs matériels complexes.

Développements : les principales contributions de cette thèse sont le développement de quatre outils accessibles en libre accès [5] [Pou20] :

1. **Un benchmark mémoire** – `DML_MEM` : ce benchmark permet de mesurer le débit mémoire en réalisant des accès dans un tableau avec des motifs de sauts (strides). La taille du tableau et des sauts peut varier et permet de caractériser les différents niveaux de la hiérarchie mémoire. Cet outil est utile pour caractériser une nouvelle architecture, mais peut aussi être utilisé lors du design d'un nouveau processeur pour vérifier le bon fonctionnement du prélecteur mémoire.
2. **Un benchmark de FPU** – `Kernel Generator` : ce générateur permet de caractériser finement la performance des unités de calcul en virgule flottante (FPU). Pour cela, il génère un benchmark assembleur en utilisant des instructions vectorielles de taille différentes (SSE, AVX2, AVX-512). L'utilisateur peut alors choisir le type d'opération à exécuter (addition, multiplication, FMA) et la précision (simple, double). La partie du benchmark mesurée ne contient que des instructions de calculs et permet de mesurer très précisément la performance atteignable par le processeur (souvent proche ou égale à la performance théorique).
3. **Un outil de suivi d'activité du bus mémoire** - `YAMB` : cet outil permet de suivre le trafic du bus mémoire. Pour cela, l'outil mesure l'activité de chaque contrôleur mémoire à l'aide de l'interface `Perf Events` en comptant le nombre de transactions (lecture et écriture) ainsi que le nombre d'accès manqués (miss) dans le dernier niveau de cache (LLC). Pour corréler l'activité du bus avec les parties du code qui en sont responsables, le graphique peut facilement être annoté en utilisant une bibliothèque C/C++/Fortran, appelée directement depuis le code source.
4. **Un outil d'analyse de performance** – `Oprofile++` : cet outil permet de désassembler le code d'une application, d'extraire les points chauds (hot spots) et d'en mesurer leur performance. Dans un premier temps, l'outil relie le profil de performance obtenu lors de l'exécution aux instructions assembleurs exécutées. Ensuite, il s'occupe d'extraire les boucles critiques et de mesurer le nombre d'instructions réalisées par cycle d'horloge (IPC). Il est ensuite possible de quantifier des opportunités d'amélioration, mais aussi de prédire la performance en fonction d'une amélioration du matériel ou du logiciel.

5. Répertoire Github regroupant les différents outils développés durant la thèse - https://github.com/PourroyJean/performance_modelisation

Publications et communications : durant cette thèse, plusieurs occasions nous ont permis de communiquer et de partager nos résultats :

1. L'outil `Kernel Generator` a fait l'objet d'une publication [PDC19] et d'une présentation dans une conférence internationale :
   Jean POURROY, *Patrick* DEMICHEL *et Denis* CHRISTOPHE. *« Assembly micro-benchmark generator for characterizing Floating Point Units ». In :* HPCS 2019 - 17th International Conference on High Performance Computing & Simulation. *Dublin, Ireland : IEEE, juil. 2019*
2. Un large état de l'art des compteurs matériels et des outils de suivi de performance à été présenté à l'occasion de la conférence `HPE TSS 2018`[6] à La Haye (Pays-Bas). Cette présentation a permis de présenter les différentes façons d'interagir avec les compteurs matériels présents sur les processeurs et de présenter un état de l'art des outils utiles pour la compréhension et l'optimisation d'une application.
3. La méthodologie et les différents outils développés durant ces travaux de thèse ont été présentés lors de la conférence `HPE TSS 2019`[6] à Paris. Cet exposé nous a permis de présenter la méthodologie mise en place permettant d'utiliser les outils développés pour faciliter la recherche de nouvelles architectures.
4. Les travaux de thèse ont fait l'objet d'une présentation lors de la conférence interne `HPE Tech Con 2020`. Sur 548 papiers seul 16 d'entre eux ont été retenus par un comité d'experts reconnu dans ce domaine. La sélection d'un papier lors de cette conférence montre l'importance des travaux présentés aux yeux de l'entreprise.

Autres évènements : lors de ces quatre années de thèse, différents évènements ont permis de partager et d'appliquer les différents outils développés :

1. Nous avons délivré une formation de deux jours à l'ENS Cachan pour une vingtaine de participants venant du monde académique, comme du monde industriel. Cet évènement fut l'occasion de présenter les premiers développements, mais aussi de partager notre méthode de travail avec des développeurs spécialisés du domaine.
2. Les outils d'analyse de performance ont été utilisés pour remporter le Hackaton du HPC 2018. Ce concours, organisé par GENCI (Grand Équipement National de Calcul Intensif), et sponsorise par Intel, avait pour objectif d'optimiser une application de calcul distribué MPI pour la dynamique de systèmes particulaires. En utilisant les outils d'analyse de performance et en appliquant les optimisations adaptées, une accélération d'un facteur 10 a pu être réalisée permettant d'obtenir le prix de la meilleure optimisation [Pou+18].

6. Hewlett Packard Enterprise Technology & Solutions Summit (HPE TSS) est l'événement annuel de Hewlett Packard Enterprise le plus important et le plus complet en matière de transfert de connaissances techniques et de solutions pour les communautés techniques HPE et partenaires d'Europe, du Moyen-Orient et d'Afrique - https://h41382.www4.hpe.com/tss/application/assets/pdf/TSS.WhyAttendPartner.pdf

1.3.2 Plan du manuscrit

Cette thèse propose une analyse étendue du domaine du HPC pour comprendre et contribuer à l'élaboration des prochaines générations de plateformes de calculs. Pour cela, le manuscrit suit la structure suivante :

— Le chapitre 2 introduit le domaine du calcul haute performance en étudiant ses origines et en présentant leur architecture actuelle et les moyens employés pour les programmer. Nous discutons ensuite de leur performance en étudiant l'évolution du classement du Top500 et nous expliquons les principaux freins qui empêchent de construire des plateformes toujours plus puissantes avec les méthodes actuelles consistant à ajouter indéfiniment des serveurs. Afin d'y parvenir, nous présentons les principales opportunités technologiques actuellement développées qui vont nous permettre de repenser l'architecture des plateformes. La fin de ce chapitre s'intéresse plus précisément à la caractérisation des microarchitectures et à l'analyse de performance d'applications.

— Le chapitre 3 présente les principaux outils développés durant ces travaux de thèse. Nous discutons des motivations et des critères de développement qui nous ont conduits à développer deux benchmarks et deux outils d'analyse de performance.

— Le chapitre 4 présente la méthodologie élaborée et utilisée pour caractériser et choisir une architecture pour une application donnée. Nous utilisons l'analyse du benchmark `STREAM` et d'un processeur Intel Skylake pour illustrer chacune des 5 étapes.

Afin de permettre la réalisation des différents codes développés, l'étude approfondie des différentes évolutions technologiques des processeurs et des techniques de suivi de performances sont présentées dans deux annexes :

— L'Annexe A couvre l'origine et l'évolution de la microarchitecture des processeurs. Nous présentons les fonctionnalités clés des processeurs modernes qu'il est nécessaire de connaître pour comprendre la performance des applications : *pipeline*, instructions vectorielles, etc. Nous étudions plus précisément la hiérarchie mémoire qui est la ressource critique de nombreuses applications.

— L'Annexe B présente les compteurs matériels. Ces registres spéciaux du processeur permettent de suivre l'exécution d'une application en mesurant le nombre d'évènements logiciels et matériels. La programmation des compteurs est très difficile et nécessite le recours à des codes bas niveaux. Plusieurs méthodes peuvent alors être employées et demandent une bonne expérience pour être mises en oeuvre.

Nous faisons référence à ces deux annexes lorsque les concepts qui y sont étudiés sont utilisés dans le manuscrit.

Chapitre 2

Calcul Haute Performance

Ce chapitre présente un état de l'art du domaine du HPC afin de motiver la nécessité de repenser l'architecture des supercalculateurs, mais aussi d'utiliser de nouvelles technologies. Pour comprendre quels challenges doivent être relevés, nous étudions le classement du Top500 et synthétisons les principaux freins au développement des performances des plateformes de calculs. Nous présentons ensuite les principales technologies qui devront être utilisées pour parvenir à développer les prochaines générations de supercalculateurs. Le chapitre se termine par l'étude des outils permettant la caractérisation et le suivi de performance des applications.

Sommaire

2.1 Introduction au Calcul Haute Performance (HPC) 16

2.1.1 Le calcul scientifique et la simulation numérique 16

2.1.2 Le calcul haute performance 18

2.1.3 Programmation parallèle et performance des supercalculateurs 25

2.1.4 Performance de la parallélisation 32

2.2 Évolution de la performance des supercalculateurs 34

2.2.1 Comparer la performance des supercalculateurs 34

2.2.2 Évolution des performances des supercalculateurs 37

2.2.3 Le futur du HPC 42

2.2.4 Défis à relever pour l'élaboration d'une plateforme exascale 47

2.3 Opportunités pour le développement de nouveaux supercalculateurs 58

2.3.1 Investissements financiers 58

2.3.2 Nouvelles technologies mémoire 60

2.3.3 Nouvelles technologies d'interconnexion. 69

2.3.4 Nouvelles architectures 72

2.3.5 Gen-Z 78

2.4 Caractérisation et analyse de performance des architectures 86

2.4.1 Caractérisation des architectures 87

2.4.2 Analyse de performance d'une application 94

2.5 Conclusion 110

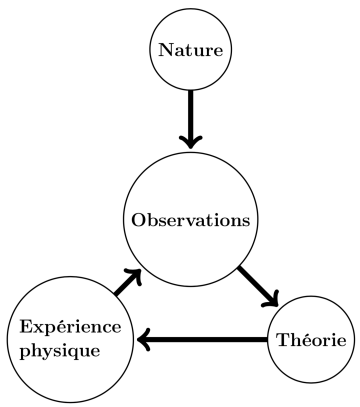
2.1 Introduction au Calcul Haute Performance (HPC)

2.1.1 Le calcul scientifique et la simulation numérique

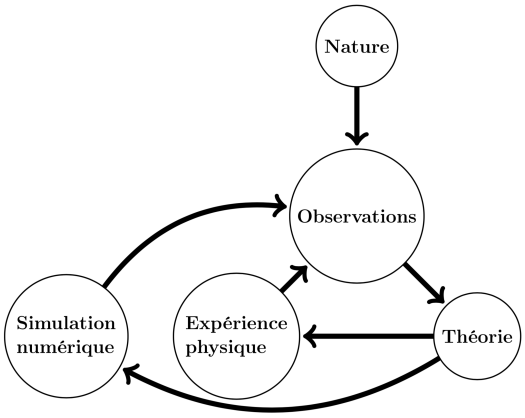
À l'origine, les scientifiques observaient la nature et émettaient des théories pour expliquer leurs observations (voir figure 2.1a). En se basant sur ces théories, ils réalisaient des expériences physiques pour les valider ou non. Ils faisaient alors de nouvelles expériences pour affiner leur théorie. Les simulations numériques sont alors apparues comme des alternatives aux expériences physiques qui étaient souvent longues et onéreuses (voir figure 2.1b). Des scientifiques comme Pythagore réalisaient ces simulations en faisant des calculs manuels ou en s'aidant de tables précalculées. Du fait de la limitation de leur capacité de calcul et du temps disponible, c'est avec l'apparition de l'informatique que les simulations numériques ont connu leur véritable essor.

2.1.1.1 Principes de la simulation numérique

La majorité des simulations numériques sont basées sur des équations dites *gouvernantes* qui sont des approximations des phénomènes étudiés. Ces équations ont besoin d'être discrétisées pour pouvoir être exécutées par un ordinateur. En mathématiques, la discrétisation est un procédé qui permet de passer d'un modèle à son équivalent discret. Ce procédé ne permet pas de décrire le phénomène réel, mais de l'approcher avec plus ou moins d'erreurs. Pour améliorer ces simulations, ces représentations doivent utiliser des maillages les plus fins possible. La figure 2.2 montre comment Météo-France découpe une région à l'aide de mailles mesurant 2,5 km de côté. Une autre approche utilise des modèles probabilistes pour représenter un comportement. Cette approche est adaptée pour des phénomènes où chaque élément peut subir différents événements. Pour chaque étape du calcul, le résultat évolue grâce à des tirages aléatoires (méthode de Monte-Carlo [Kro+14]). Pour améliorer la précision des simulations, il est donc nécessaire d'augmenter



(a) Les expériences physiques permettent de valider les théories.



(b) La simulation numérique peut être une alternative aux expériences physiques.

FIGURE 2.1 – La simulation numérique a apporté une nouvelle façon d'expérimenter les théories.

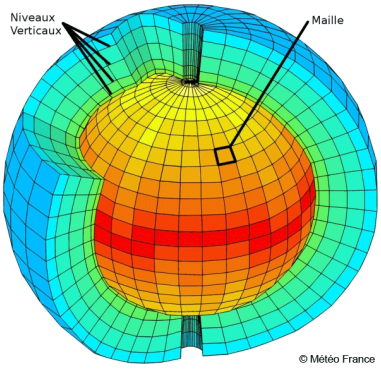


FIGURE 2.2 – Le maillage le plus fin exploité par Météo-France [1]pour ses prévisions régionales restitue des mailles de 2,5 km de côté.

le nombre de tirages et donc le nombre de calculs à réaliser. Ainsi, que ce soit pour affiner la taille des maillages ou augmenter le nombre de tirages, les simulations numériques nécessitent de grandes puissances de calcul.

2.1.1.2 Quelques applications de la simulation numérique

Lorsque nous évoquons la simulation numérique, nous pensons souvent aux domaines physiques (météorologie, mécanique, biologie). Cependant, elle est aussi largement utilisée dans d'autres domaines comme en sciences humaines (sociologie, analyse démographique) ou dans le domaine de la sécurité nationale. En effet, Alan Turing a développé les premiers ordinateurs lors de la fin de la Deuxième Guerre mondiale pour aider au décryptage des messages allemands. En 1936, il avait présenté les premiers concepts de programmes et donné naissance à l'aide d'une expérience de pensée à l'ancêtre des ordinateurs nommé machine de Turing. L'utilisation de simulations numériques a de nombreux avantages. En plus de profiter de la puissance de calcul des ordinateurs, elle permet de simuler des phénomènes dont les conditions ne sont pas reproductibles sur terre (physique théorique). Un autre avantage est de réduire drastiquement les coûts d'une expérience, par exemple pour la réalisation de crash automobile, où ce ne sont plus de réels modèles de voiture, mais bien des voitures virtuelles qui sont *crashées* sur des murs.

Dans le domaine de la santé, l'étude de la structure des protéines est primordiale. Ces molécules qui assurent les fonctions élémentaires d'une cellule interviennent dans la majorité des processus biologiques (régulation du métabolisme, défense immunitaire). Dans ce genre de simulation, le pas de temps est de l'ordre de la nanoseconde. Grâce à la simulation numérique, une meilleure compréhension de ces molécules permet la découverte de nouveaux médicaments ou antibiotiques. Aussi, des modélisations peuvent être utilisées pour analyser la propagation d'un virus comme la grippe aviaire à l'échelle mondiale pour mieux protéger les populations[CEA07]. En février 2020, le gouvernement français indiquait travailler sur la modélisation de la propa-

1. Source du graphique : http://www.irma-grenoble.com/

gation du COVID-19 [2] (coronavirus). Cette modélisation fait intervenir plusieurs paramètres comme le lieu et la période d'incubation du virus ou encore la fréquentation et les destinations des passagers des 4000 principaux aéroports mondiaux. En mars 2020, le supercalculateur le plus puissant au monde (Summit) était utilisé pour identifier 77 molécules potentiellement efficaces contre la [3] COVID-19 [SS20].

En astrophysique, la simulation numérique est aussi capitale du fait de la non-reproductibilité des expériences en laboratoire. À l'opposé de l'exemple précédent sur les protéines, elle permet d'étudier des objets aussi grands que complexes comme le système solaire, les galaxies ou bien l'univers. La simulation permet alors de faire évoluer le système avec un pas de temps allant jusqu'au million d'années. Une équipe du CEA a pu reconstituer le passé de la galaxie Andromède en analysant les observations réalisées grâce au satellite infrarouge Spitzer [Blo+06]. Après avoir mis au point le modèle adéquat et après plusieurs heures de simulation, ils ont pu déterminer qu'elle avait été percutée par une galaxie voisine il y a plus de 210 millions d'années et que sa forme actuelle résultait de cet impact (voir figure 2.3).

Pour la compétitivité et l'innovation des entreprises, les simulations numériques sont désormais un outil indispensable. Cet outil les aide à la conception, à la décision et au contrôle de leurs activités. Dans l'industrie des hydrocarbures, la recherche pétrolière est très coûteuse : le coût d'un forage d'exploration maritime peut atteindre 100 millions d'euros [5]. Les pétroliers utilisent la simulation numérique pour analyser les fonds marins, modéliser les réservoirs de pétrole et, in fine, optimiser l'extraction du pétrole. Pour cela, ils utilisent des bateaux tractant d'immenses lignes flottantes pourvues de capteurs. La figure 2.4 illustre comment de tels bateaux sont utilisés pour émettre des explosions, et analyser le réfléchissement des signaux sur le fond marin. Grâce à l'analyse de ces données, il est alors possible de construire une cartographie du fond marin et ainsi déceler la présence ou non de réservoirs d'hydrocarbures. En perçant le puits de façon optimale, il sera d'autant mieux exploité. De plus, le risque de forer au mauvais endroit est lui aussi diminué. Actuellement, le taux de succès est de deux forages sur trois.

2.1.2 Le calcul haute performance

Aujourd'hui, presque que tout ce que nous utilisons a été simulé, à un tel point que l'avancée de nos sociétés dépend de la puissance de calcul disponible pour réaliser ces simulations. La rapidité de l'exécution des simulations numériques dépend alors de la puissance de calcul disponible pour exécuter l'application. Le domaine du Calcul Haute Performance (HPC) est le domaine informatique qui consiste à exécuter ces applications le plus efficacement possible sur une plateforme. Celle-ci peut être un simple ordinateur personnel ou un centre de calculs dédié. De telles infrastructures sont utilisées dans de nombreux domaines dont les principaux sont la simulation numérique, la cryptographie, l'analyse de données (*big data*), l'intelligence artificielle

2. https://www.gouvernement.fr/info-coronavirus
3. http://www.academie-francaise.fr/le-covid-19-ou-la-covid-19
4. Image et légende extraites de http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast.php?id_ast=958&t=actu
5. https://www.planete-energies.com/fr/medias/decryptages/la-difficile-decision-de-lancer-un-forage

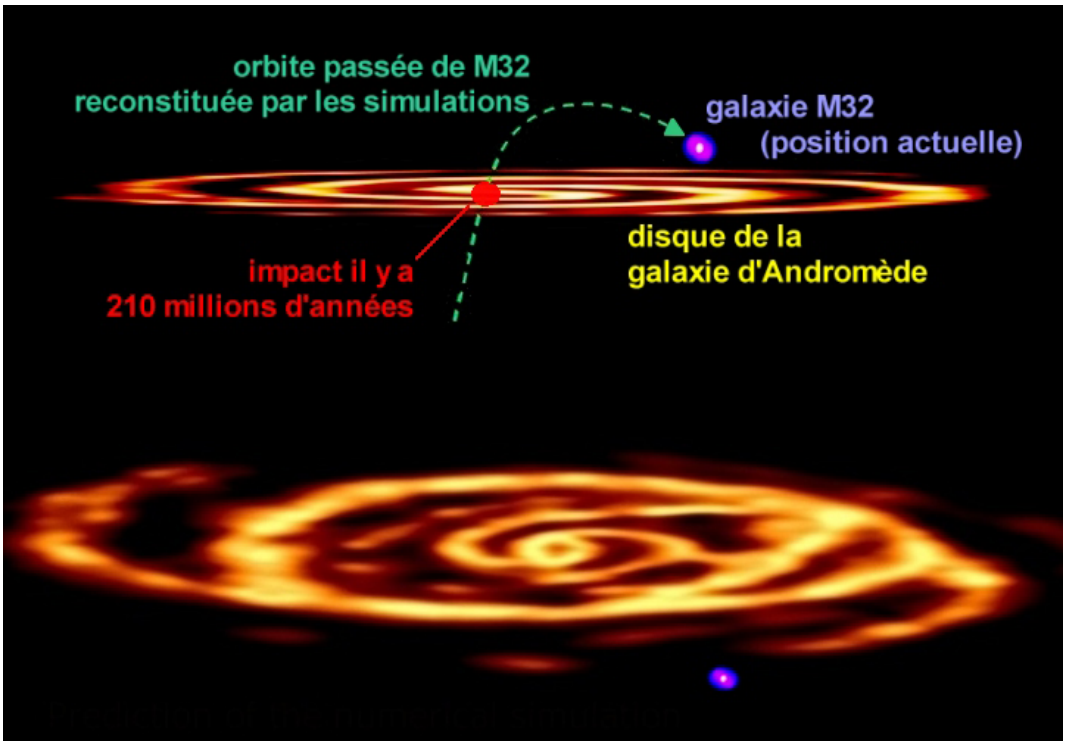


FIGURE 2.3 – Reconstitution de l'évolution passée de la galaxie d'Andromède à l'aide de simulations numériques. En haut : orbite passée de la galaxie M32 à travers le disque de la galaxie d'Andromède (vu ici par la tranche) et localisation de l'impact de la collision frontale entre les deux galaxies. En bas : résultat de la simulation reproduisant l'émission infrarouge actuelle, notamment la structure en double-anneau et la morphologie particulière de l'anneau externe [4].

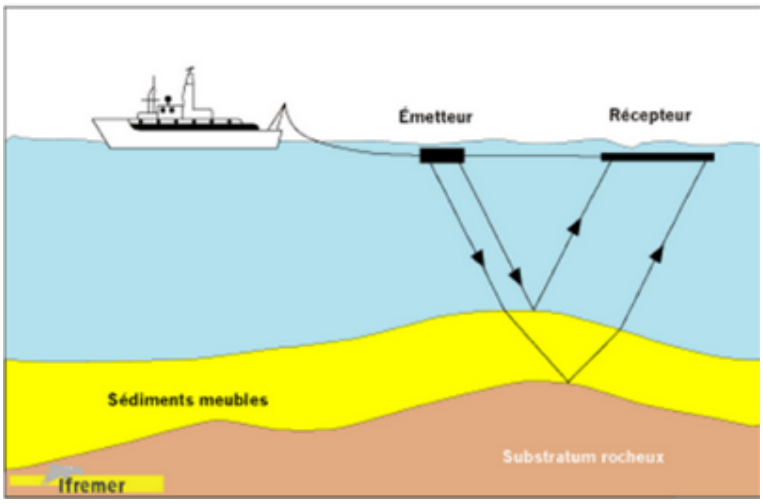


FIGURE 2.4 – Étude de fonds marins grâce à la sismique de réflexion.

ou la sécurité. En fonction des besoins des applications (puissance de calcul, espace mémoire, stockage), mais aussi des budgets alloués, des infrastructures adaptées sont développées :

1. **Le supercalculateur dédié** (*dedicated supercomputer*) consiste à la création d'une architecture unique qui ne sera pas répliquée. Le premier supercalculateur élaboré par Seymour Cray (le CDC6600), alors employé de l'entreprise Control Data, utilisait un tel mode de conception. Il était alors l'ordinateur le plus puissant de la planète grâce à une mémoire de 131 000 mots de 60 bits, une fréquence de 40 MHz et sa capacité calculatoire de 3.3×10^6 opérations par seconde. À titre de comparaison, l'ordinateur portable utilisé pour écrire ce manuscrit est capable d'exécuter plus de 10^9 opérations par seconde. L'objectif de ce type d'infrastructure est d'être parfaitement adaptée aux besoins de l'application. La conception de ces architectures étant unique, les frais de conception sont très élevés, mais cette spécificité en fait des architectures très performantes.
2. **Le calculateur à matériel de base** (*commodity cluster*). L'augmentation de la vitesse des circuits ainsi que l'augmentation exponentielle du nombre de transistors des circuits a rendu les supercalculateurs dédiés très difficiles à alimenter en énergie ou à être refroidit. Il a alors fallu repenser leur architecture pour continuer à augmenter la puissance des supercalculateurs. Les *commodity cluster* agrègent du matériel grand public (haut de gamme) pour former des grappes de calculs de plusieurs milliers de processeurs. La performance de ces architectures ne repose pas sur l'utilisation de matériels ultra-optimisés, mais sur l'agrégation de milliers de serveurs (noeuds de calculs) travaillant ensemble. L'Intel Paragon est un des premiers exemples de cluster construit par Intel en 1992 [6], qui regroupe 3680 processeurs indépendants atteignant une puissance cumulée de 143.40×10^9 opérations à virgule flottante (FLOP). Ce modèle de supercalculateur est le plus répandu actuellement et a permis l'élaboration des supercalculateurs les plus puissants de la planète. Ces infrastructures sont généralement développées pour des industries privées ou de grandes universités (voir figure 2.5).
3. **L'informatique en nuage** (*cloud computing*), utilise le modèle *System as a Service* (SAS) pour apporter aux entreprises manquant de moyens (financier, humain) ou de compétences, un accès à une infrastructure HPC externalisée. Les principaux avantages sont la flexibilité d'usage (adapter l'infrastructure à son besoin) ainsi que la sous-traitance de la gestion de la plateforme. L'informatique en nuage permet à de petites structures (PME, petites universités) d'avoir accès à une ressource de calculs facilement à coûts modérés. Nous pouvons notamment citer le projet SIMSEO [Sag16] qui a pour but de sensibiliser et d'accompagner les PME à utiliser la simulation numérique.
4. **Les grilles informatiques** (*Grid Computing*) sont un regroupement de ressources informatiques à grande échelle (nationale ou internationale). Par exemple *Einstein@Home* [Abb+09] est un projet de recherche mondial sur les ondes gravitationnelles qui regroupe les ordinateurs de 50000 utilisateurs connectés à travers le monde pour analyser les don-

6. source : https://www.top500.org/featured/systems/intel-xps-140-paragon-sandia-national-labs/#Historical

FIGURE 2.5 – Le supercalculateur du Centre de Calcul de Barcelone (BSC) est installé dans une ancienne chapelle.

nées transcrites par des capteurs. En octobre 2000, l'université de Stanford a déployé un projet de calcul distribué, nommé *Folding@Home* [Lar+09], permettant simuler la dynamique des protéines (pliage et mouvement) impliquées dans une grande variété de maladies. Pour contribuer aux calculs, il est possible d'installer un logiciel sur son ordinateur qui effectue les calculs lorsque ce dernier n'est pas utilisé. En avril 2020, alors que plus de la moitié de la population est confinée chez elle à cause de l'épidémie du COVID-19, la puissance de calcul du projet *Folding@Home* est utilisée pour étudier le coronavirus SARS-CoV-2 et aider au développement d'un vaccin [7]. Le réseau compte plus de 2 millions de processeurs et 700 000 GPU, formant la plateforme de calcul la plus puissante de la planète ($2,5 \times 10^{18}$ opérations par seconde).

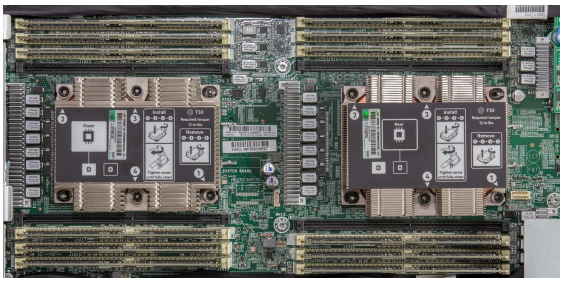
2.1.2.1 Architecture des supercalculateurs

Les deux principales caractéristiques d'un supercalculateur sont sa capacité à calculer rapidement et à mémoriser les informations (données à traiter, résultats produits). L'architecture des supercalculateurs modernes n'est pas très différente de celle des ordinateurs classiques. La puissance d'une telle plateforme vient principalement de l'agrégation de centaines de ressources de calculs, capables de travailler ensemble pour résoudre un problème complexe. Le domaine du HPC consiste à exécuter efficacement des applications sur de telles architectures. Pour cela, l'infrastructure utilisée doit gérer l'équilibrage de la charge de calcul sur les différentes ressources

7. Coronavirus : comment votre ordinateur pourrait aider à trouver un vaccin - https://www.ft.com/video/51bc9532-b08d-4129-b4db-c88ca5d42342

disponibles ainsi que la gestion des communications entre les ressources. Un supercalculateur moderne possède généralement les cinq parties suivantes :

1. **Les serveurs** utilisés dans un supercalculateur possèdent un ou plusieurs processeurs. Comme un ordinateur grand public, chaque processeur est relié à la mémoire et au stockage (voir figure 2.6a). En fonction des configurations, chaque serveur peut être agrémenté d'un ou plusieurs accélérateurs (GPU). La principale différence avec un ordinateur classique, autre que la puissance des matériels utilisés, vient du système d'interconnexion (voir figure 2.6b). Les serveurs doivent pouvoir échanger des informations pour se synchroniser ou partager des résultats entre eux et accéder au stockage (voir point 4). Dans la suite du manuscrit, nous faisons référence à un serveur à l'aide des termes "noeud" ou "noeud de calcul".



(a) Carte mère d'un serveur possédant deux processeurs.

(b) Vu arrière d'un serveur exposant les différentes interfaces de connexion.

FIGURE 2.6 – Exemple d'un serveur utilisé dans les supercalculateurs.

2. **Les accélérateurs**. Un principe fondamental des processeurs énoncé par Von Neumann est leur universalité (l'architecture Von Neumann est présentée dans l'Annexe A.2.1.2). Leur faculté à exécuter tout type de code est à la fois une force, mais aussi une de leur plus grande faiblesse. Contrairement à l'architecture Harvard (présentée dans l'Annexe A.2.1.3), les processeurs Von Neumann sont très peu efficaces (en temps et en énergie) pour résoudre certains types de calculs tels que les traitements de signaux ou d'objets géométriques. Pour accélérer certaines classes d'applications, des architectures spécialisées, appelées accélérateurs, sont utilisées pour améliorer la performance et la consommation énergétique des infrastructures :

   :

   — **Les GPU**. À l'origine, les processeurs graphiques (GPU) étaient destinés au domaine des jeux vidéo et aux calculs de rendus graphiques. Suite à leur utilisation

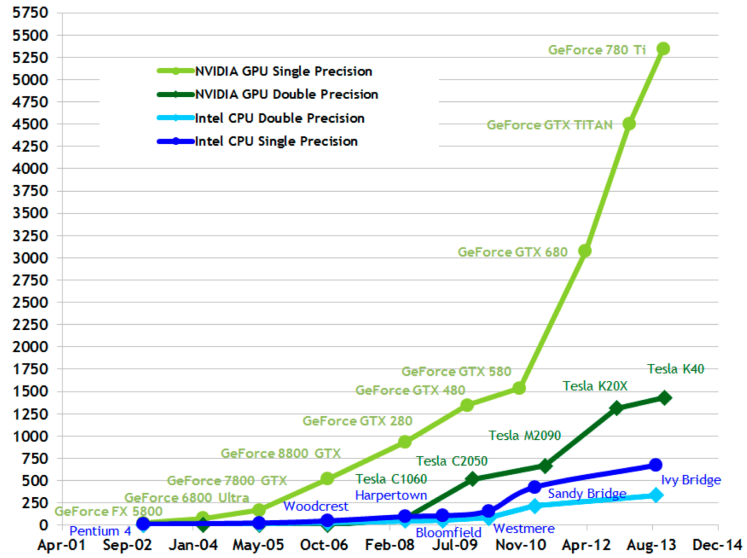


FIGURE 2.7 – Comparaison des performances entre différentes générations de CPU et de GPU

pour d'autres applications que le rendu graphique, ces accélérateurs sont aussi nommés GP-GPU pour *general purpose GPU*. À l'inverse des processeurs, les GPU sont composés de nombreux coeurs (plusieurs centaines). Cs coeurs sont moins complexes et performants que les coeurs de processeur, mais leur nombre les rend particulièrement efficaces pour certaines applications. Aujourd'hui, les applications d'intelligence artificielle, notamment celles utilisant des réseaux de neurones, sont exécutées à l'aide de GPU pour accélérer l'entraînement des modèles. La figure 2.7 compare l'évolution des performances des GPU avec celles des CPU de même génération. On remarque l'écart significatif séparant leur performance. Cependant, la figure représente la performance théorique des architectures et il est rare que les applications s'en approchent (les principales raisons sont présentées dans la section 2.2.2.2). Actuellement, il existe deux fournisseurs principaux de GPU (Nvidia et AMD) qui ont pu bénéficier de leur longue expérience dans le domaine du jeu vidéo. La première nécessite l'utilisation de langage propriétaire (`CUDA`) quand la deuxième peut être programmée grâce au langage `OpenCL`. Le principal avantage de `CUDA` est de pouvoir utiliser des librairies optimisées par NVIDIA pour ses plateformes pour différents domaines (algèbre linéaire (cuBLAS, CUSPARSE), analyse de signal (cuFFT), apprentissage machine (cuDNN, TensorFlow). `OpenCL` a été initié par Apple et est maintenant maintenu par le groupe Khronos. C'est un standard ouvert et libre de droits permettant l'exécution de codes sur différentes plateformes. Une application utilisant `OpenCL` pourra être facilement portée sur d'autres architectures (CPU, FPGA, DSP).

— **Les FPGA.** Les réseaux logiques programmables (FPGA) sont de simples circuits intégrés qui contiennent des portes logiques pouvant être programmées pour la réalisation d'un circuit. Les méthodes traditionnelles de développement sur FPGA néces-

sitent la conception d'une architecture matérielle utilisant un langage adapté comme le Hardware Description Language (HDL). Pour ce faire, le développeur doit concevoir les chemins de données ainsi que la logique de contrôle. Il s'agit d'un processus de développement fastidieux et complexe qui peut entraver leur adoption. Les FPGA les plus récents peuvent être programmés à l'aide d'`OpenCL`. Implémenter un circuit à l'aide d'un FPGA peut fournir une efficacité énergétique légèrement inférieure à celle obtenue par une implémentation d'un circuit intégré propre à une application (ASIC). Cependant, le fait que les FPGA soient programmables leur donne un avantage sur les ASIC, car le même matériel peut être reprogrammé en un nouveau circuit. Le temps de reprogrammation étant relativement faible, il peut être intéressant de reprogrammer le FPGA plusieurs fois pour les différentes phases de l'application. Cependant, la complexité de programmation des FPGA et leur prix très élevé sont les principaux freins pour leur adoption dans les supercalculateurs.

— **Les DSP**. Les processeur de signal numérique (DSP) firstplurals sont adaptés pour les opérations de convolution et de filtrage très utilisés par les algorithmes d'analyse de signal. Généralement contraints par leur consommation électrique (matériel embarqué, smartphone, robotique), les DSP utilisent des fréquences plus basses que les processeurs. Du fait de leur implémentation d'opérations optimisées, elles sont plus efficaces que les architectures standards pour ces applications. Les opérations de filtrage nécessitent, pour être optimales, d'avoir accès à chaque cycle à une instruction et deux opérandes. Les architectures Harvard utilisées par les DSP (voir Annexe A.2.1.3) sont alors plus optimales pour ce type d'exécution grâce à leurs deux bus d'accès (données et instructions).

3. **Le stockage**. Les applications de calculs intensifs ont besoin d'avoir accès à une grande capacité de stockage afin de pouvoir stocker les données à traiter et les résultats produits. Certaines applications utilisent des jeux de données dépassant de plusieurs facteurs la capacité de stockage d'un serveur (un noeud). Ceux-ci doivent donc être stockés à l'extérieur du serveur (baies de stockage). D'autres applications peuvent produire des résultats ne pouvant pas non plus être stockés sur les serveurs de calculs. Pour ces deux utilisations, un supercalculateur doit posséder un stockage avec les meilleures caractéristiques possible : latences, débits, fiabilité. Suivant le type d'application, la priorité n'est pas la même.

4. **L'interconnexion** L'exécution des applications (et les données traitées) étant réparties sur plusieurs serveurs, ces derniers ont besoin d'être interconnectés pour s'échanger des informations (résultats temporaires, instructions) ou pour se synchroniser. Le système d'interconnexion participe à une part importante de la performance finale. Le matériel utilisé doit donc être adapté aux besoins de l'application. Suivant sa nature, une application bénéficie d'un réseau avec une faible latence et/ou un débit élevé. La latence inclut le temps nécessaire aux routeurs pour rediriger le message au destinataire. Pour cela, la structure des routeurs doit être bien pensée pour réduire le nombre de *sauts* de chaque échange, c'est-à-dire le nombre d'intermédiaires pour aller de l'expéditeur au

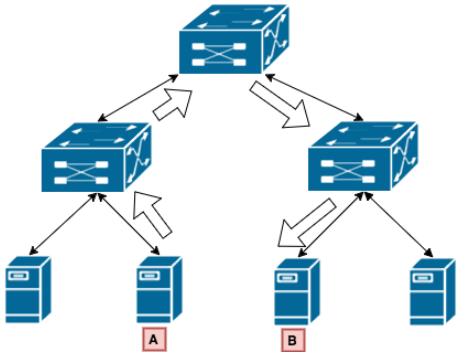


FIGURE 2.8 – Exemple d'une topologie d'un cluster. Si le serveur A veut envoyer un message à B, il devra effectuer 4 sauts (ou *hop*).

destinataire. La figure 2.8 montre un exemple de topologie et l'impact qu'elle a sur la latence des communications qui peut être multipliée par deux suivant le destinataire du noeud A. Cette structure (ou topologie) doit s'assurer d'éviter tous risques de congestions afin qu'une partie de réseau ne soit pas trop sollicitée ce qui créerait des baisses de performances. Différentes technologies sont ainsi développées pour répondre aux différents besoins (et différents budgets). Les technologies dites `10GbE` (*10 Gigabit Ethernet*), basées sur TCP, permettent d'échanger des informations pour un débit allant jusqu'à 10 Gb/s. Ce réseau peut être implémenté à l'aide de câbles en cuivre ou de fibres optiques. Un second protocole appelé Infiniband permet d'atteindre des débits plus élevés tout en assurant la réussite des transferts. En fonction de l'application utilisée, le choix de la technologie réseau utilisée peut avoir un fort impact sur ses performances [Cou09]. Aussi, l'efficacité du réseau ne dépend pas seulement du matériel, la partie logiciel est tout aussi importante comme la complexité des protocoles ou la performance des algorithmes de routage.

5. **Le système de refroidissement**. La puissance électrique utilisée par les différents matériels est dissipée sous forme de chaleur. Il n'est pas rare qu'une armoire (un *rack*) consomme à elle toute seule plus de 50 kW. Un supercalculateur doit donc posséder un système de refroidissement performant pour éviter les surchauffes des composants. En effet, celles-ci peuvent provoquer un ralentissement des matériels (fréquences des processeurs) ou même les détériorer. De nombreuses technologies ont été développées pour améliorer le rendement des systèmes de refroidissement tels que le refroidissement liquide ou l'immersion des composants dans des bains d'huile.

2.1.3 Programmation parallèle et performance des supercalculateurs

2.1.3.1 Le calcul parallèle

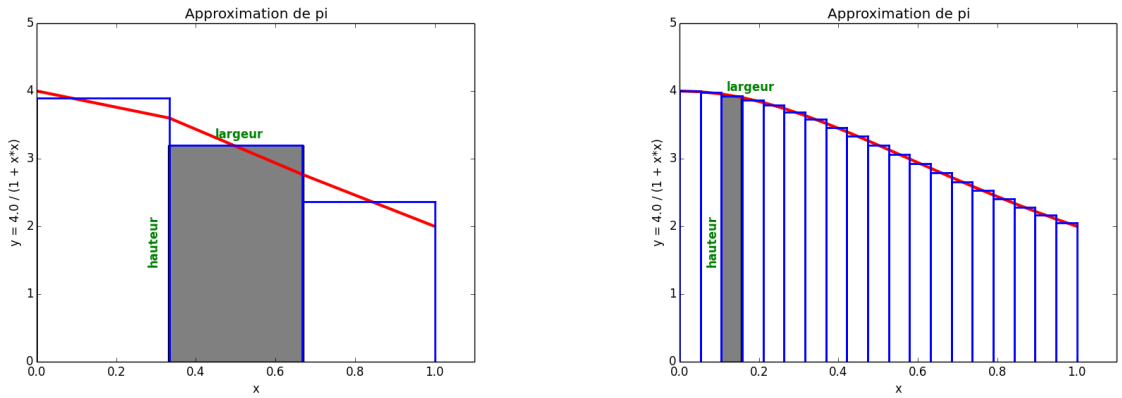
L'agrégation de milliers de ressources de calcul (processeurs, accélérateurs) a pour objectif d'accélérer l'exécution d'applications qui ne pourraient pas être réalisées par une seule d'entre

elles dans un temps raisonnable. Le travail des programmeurs est alors de diviser ce problème complexe en sous-problèmes indépendants, pouvant être résolus simultanément par les différentes ressources. Il faut ensuite regrouper les différents résultats intermédiaires pour calculer le résultat final. Cette méthode de résolution, appelée calcul parallèle, regroupe l'ensemble de moyens, logiciels et matériels qui permettent de réduire le temps de calcul d'un programme en répartissant la charge de travail.

Pour illustrer la nécessité d'avoir recours aux techniques de calcul parallèle, nous présentons un exemple simple permettant de réaliser l'approximation de π par le calcul de l'intégrale suivante :

$$\int_0^1 \frac{4.0}{1+x^2} \tag{2.1}$$

Calculer une intégrale revient à calculer l'aire formée par cette courbe et l'axe des abscisses dans le domaine étudié, ici $[0, 1]$. Cette surface peut être approximée par la méthode des rectangles (ou trapèzes). Cette méthode revient à dessiner des rectangles sous la courbe (voir figure 2.9). La surface de ces derniers pouvant être calculée, il est alors possible d'approcher l'intégrale donnée dans l'équation 2.1 à l'aide du code présenté dans l'extrait 2.1.



(a) Approximation avec 4 rectangles (b) Approximation avec 14 rectangles

FIGURE 2.9 – Méthode des rectangles : exemple de deux exécutions de l'algorithme avec un nombre de rectangles différents.

Nous remarquons sur la figure 2.9a, qu'avec l'utilisation de 4 rectangles, la surface des rectangles ne correspond pas exactement à la courbe à approcher. Ces approximations affectent la précision de notre programme qui affiche une valeur de $\pi = 3.38$. La figure 2.9b présente le même programme qui utilise 14 rectangles. Les rectangles sont alors plus fidèles à la courbe et le programme calcule une valeur de $\pi = 3.21$. Nous présentons cet exemple simpliste pour illustrer le besoin de puissance de calcul pour réaliser des simulations précises :

— La précision du calcul dépend du nombre de rectangles choisis. Plus le nombre de rectangles utilisés est grand, plus la précision augmente. Cependant, augmenter le nombre de rectangles à calculer augmente le nombre d'opérations nécessaires à réaliser. L'aug-

```
double x, largeur, hauteur, pi = 0.0;

int num_steps = 4;
int num_steps = 14;

largeur = 1.0 / num_steps;

for (int i = 0; i < num_steps; ++i) {
    x =  (i) * largeur;
    hauteur = ( 4.0/(1.0+x*x));
    pi += largeur * hauteur;
}

cout << "Valeur de pi: " << pi << endl;
```

Extrait 2.1 – Implémentation de l'algorithme de calcul d'intégrale par la méthode des rectangles

mentation déraisonnable du nombre de rectangles utilisés apporte aussi des imprécisions de calculs liées à l'arithmétique flottante utilisant des registres de taille finie.

— Pour accélérer le calcul de cette application, le calcul des différents rectangles doit être réparti entre les ressources de calcul. À la fin de l'exécution, un processus s'occupe de récolter et d'additionner les différents résultats.

— La capacité d'une application à utiliser des ressources parallèles n'est pas automatique. Une transformation du code doit être réalisée de manière explicite (par le programmeur) ou implicite (par le compilateur ou l'utilisation de librairies spécialisées).

2.1.3.2 Niveau de parallélisme des supercalculateurs

La performance d'une application parallèle dépend donc du nombre et de la performance des ressources pouvant travailler simultanément pour la résolution du problème. La tâche des programmeurs est d'adapter le code pour répartir l'application sur les différentes ressources disponibles. Ce travail est difficile, car le parallélisme est présent à plusieurs niveaux dans un supercalculateur.

Taxonomie de Flynn. Les différentes formes de parallélisme ont été regroupées en quatre classes appelées taxonomie de Flynn [Fly11] (voir figure 2.10). Ces classes permettent de caractériser les architectures en fonction de l'indépendance ou non des instructions et des données :

— La classe `SISD` représente les architectures classiques n'exécutant qu'une instruction sur une donnée à la fois ;

— La classe `SIMD` regroupe les architectures vectorielles exécutant une instruction sur plusieurs données à la fois (voir Annexe A.2.2.2) ;

— La classe d'architecture `MISD` exécute des instructions différentes sur le même jeu de données. Les processeurs capables d'exécuter des instructions réalisant une multiplication et une addition fusionnées (FMA), peuvent être considérés comme tels. Le processeur

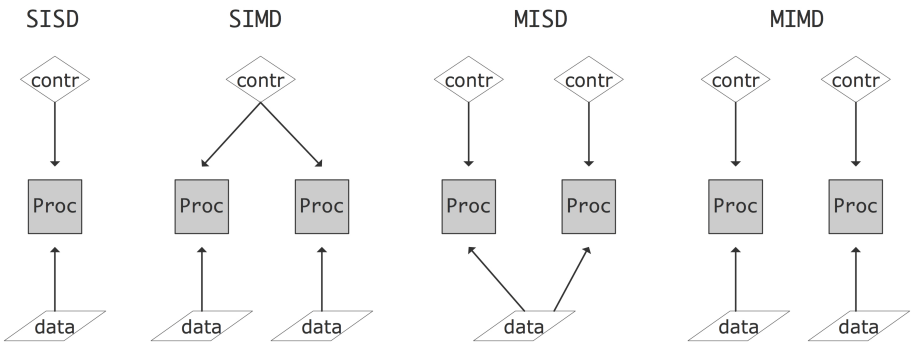


FIGURE 2.10 – Les quatre classes de la taxonomie de Flynn (graphique extrait de [Eij13])

ASIC présent dans les voitures connectées peut par exemple exécuter deux algorithmes différents sur les mêmes données, pour s'assurer de la validité d'une décision à prendre ;

— La dernière classe `MIMD` représente des architectures exécutant plusieurs instructions sur plusieurs données à la fois. Un serveur possédant plusieurs processeurs est un exemple d'architecture `MIMD`.

Niveaux de parallélisme d'un supercalculateur. Dans un supercalculateur, ces différentes formes de parallélisme peuvent être retrouvé à différents niveaux, représentés sur la figure 2.11 :

— Le niveau 1 concerne les serveurs, reliés par un système d'interconnexion. Différentes tâches peuvent être assignées à chaque noeud qui peuvent communiquer entre eux pour se synchroniser ou partager des résultats.

— Le niveau 2 concerne le parallélisme des processeurs (ainsi que des accélérateurs si le serveur en possède). En fonction des tâches à réaliser et des caractéristiques des ressources de calculs, les tâches peuvent être réparties pour être exécutées.

— Le niveau 3 est situé dans les processeurs et concerne l'utilisation de processeurs multi-coeurs.

— Le niveau 4 est lié aux processeurs superscalaires possédant plusieurs pipelines (voir Annexe A.2.4.1).

— Le niveau 5 de parallélisme concerne les unités de calcul vectorielles des processeurs qui peuvent exécuter une instruction sur plusieurs données simultanément (voir Annexe A.2.3.4).

Pour maximiser le nombre d'instructions pouvant être exécutées sur un supercalculateur (*Instruction Level Parallelism* (ILP)), différentes méthodes existent. Le niveau le plus haut consiste à exécuter des applications indépendantes permettant de saturer l'utilisation de la plateforme (*Job Level Parallelism*). Une même application peut être découpée en tâches qui peuvent être exécutées en parallèle (*Task Level Parallelism* (TLP) [Kam+09]). Lorsque la nature du code le permet, une instruction peut être exécutée sur plusieurs données simultanément (*Data Level Parallelism*(DLP) [EV97]).

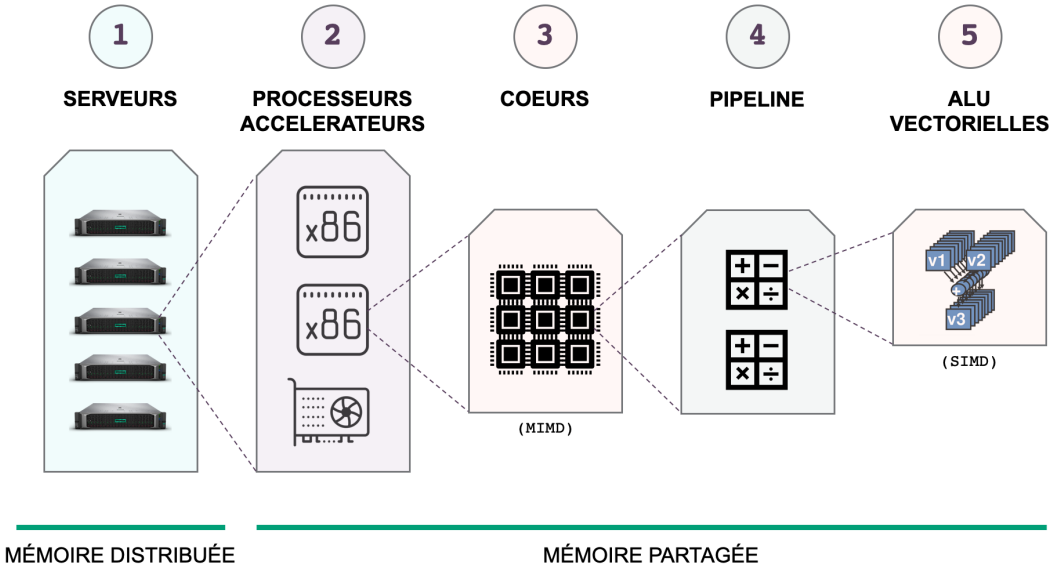


FIGURE 2.11 – Différents niveaux de parallélisme dans un supercalculateur. Les processeurs MIMD sont capables d'exécuter plusieurs instructions sur plusieurs données grâce à l'utilisation de multiples coeurs. Les unités arithmétiques et logiques (ALU) peuvent exécuter une instruction sur plusieurs données à la fois (MIMD) grâce aux instructions vectorielles.

2.1.3.3 Paradigme de programmation

Pour pouvoir bénéficier des ressources de calculs disponibles, le programmeur doit adapter certaines parties du code. Les niveaux de parallélisme les plus bas (pipeline et ALU) sont gérés par le matériel sans intervention de l'utilisateur. Par exemple, lorsque le compilateur génère des instructions vectorielles, l'unité de calcul s'occupe seule de leur exécution. Cependant, le programmeur doit connaître les détails de la microarchitecture pour développer du code pouvant profiter du parallélisme. Il peut par exemple adapter la structure de son code pour permettre au compilateur de présenter plusieurs instructions au module d'exécution dans le désordre (voir Annexe A.2.3.5) permettant ainsi d'optimiser l'utilisation des pipelines (voir Annexe A.2.4.1) présents dans les processeurs superscalaires (voir Annexe A.2.4.2).

Pour bénéficier des niveaux 1, 2 et 3 (voir figure 2.11), deux paradigmes fondamentaux de la programmation parallèle peuvent être utilisés : la programmation à mémoire partagée ou à mémoire distribuée. La principale distinction entre ces deux paradigmes est le partage ou non d'un espace mémoire entre les ressources utilisées (voir figure 2.12).

Programmation à mémoire distribuée. Le paradigme de programmation à mémoire distribuée doit être utilisé lorsque les différents processus n'ont pas accès au même espace mémoire. Les communications entre les ressources de calculs sont alors réalisées par le système d'interconnexion. L'avantage de telles architectures est de permettre d'agréger plus ou moins de ressources en fonction du besoin d'un utilisateur. La performance de la solution est alors dépendante de celle du système d'interconnexion et du nombre de serveurs utilisés.

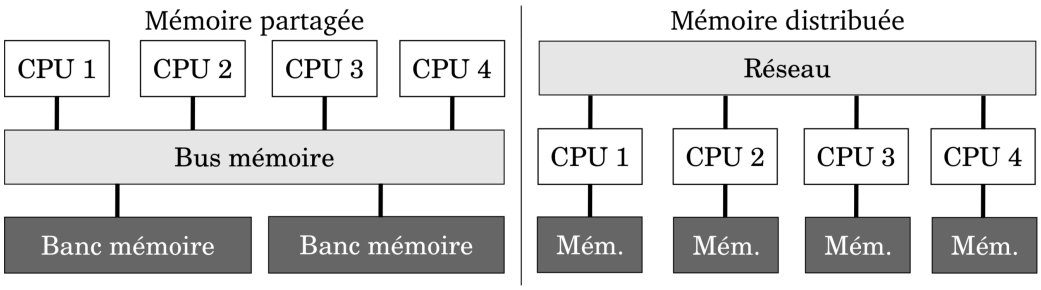


FIGURE 2.12 – En fonction de l'architecture et du mode d'accès mémoire, deux paradigmes de programmations peuvent être utilisés (graphique extrait de [Val16]).

Les données nécessaires pour les calculs ainsi que les résultats temporaires doivent être explicitement transférés par l'utilisateur. Pour cela, les programmeurs peuvent utiliser des librairies de *passage de messages*, dont le standard le plus utilisé est une interface de passage de messages (MPI). Le standard définit la sémantique des différentes fonctions, identifie les tâches distribuées et propose des moyens d'échange et de synchronisation entre les processus. Le standard est implémenté par des librairies telles que `OpenMPI`, `MPICH-2` ou encore `IntelMPI`. Chaque noeud ayant son propre système d'exploitation, les librairies doivent être installées sur tous les serveurs utilisés.

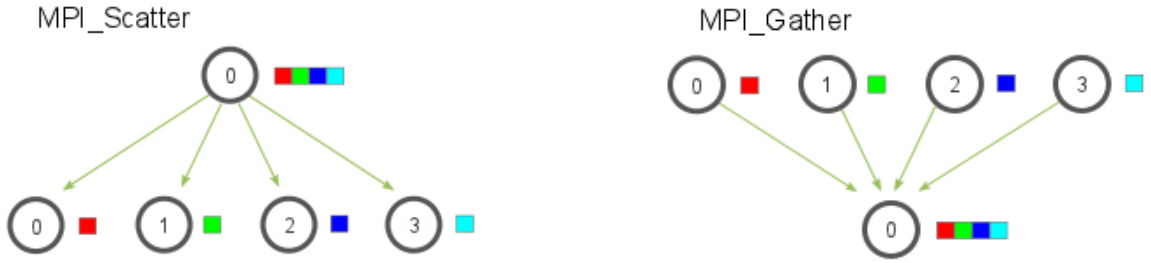
La principale difficulté de ce paradigme de programmation est la nécessité de réaliser explicitement les mouvements de données entre les noeuds. L'exécution d'une application peut alors être résumée en 3 étapes. La figure 2.13 présente les étapes 1 et 3 d'un programme de calcul à mémoire distribuée :

1. La première étape consiste à répartir le jeu de données à l'aide d'opérations de type *scatter* (voir figure 2.13a).
2. La deuxième étape est la réalisation du calcul par chaque ressource. Cette étape utilise le paradigme de programmation à mémoire partagée (voir ci-dessous).
3. Lorsque chacune d'entre elles a terminé son calcul, des opérations de type *gather* (voir figure 2.13b) permettent de récolter l'ensemble des résultats partiels, pour calculer le résultat final.

Les étapes 1 et 3 utilisent le système d'interconnexion du supercalculateur. La performance de celui-ci peut fortement impacter celle des applications. Cette particularité est à l'origine de nombreuses erreurs et rend la programmation à mémoire distribuée difficile. Le programmeur doit avoir à sa disposition des outils lui permettant de suivre l'évolution de chaque phase pour déceler des problèmes de synchronisation ou de déséquilibre de charge.

Programmation à mémoire partagée. Lorsque toutes les ressources de calculs ont accès au même espace mémoire, il est conseillé d'utiliser le paradigme de programmation à mémoire

8. Source des graphiques - https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/



(a) Les opérations de *scatter* permettent de répartir un jeu de données entre les ressources.

(b) Chaque ressource renvoie son résultat local pour calculer le résultat final du calcul.

FIGURE 2.13 – Les deux opérations principales de la programmation à mémoire distribuée sont de répartir (a) et de récolter (b) les données utilisées par les différentes ressources participant à la résolution [8].

partagée. Comme il n'est plus utile de transférer les données explicitement entre les ressources de calcul, il est possible d'utiliser des processus légers (threads). Pour profiter des niveaux de parallélisme 2 et 3 (voir figure 2.11) les programmeurs peuvent avoir recours à des librairies ou même des langages spécifiques aux processeurs ciblés. Pour les processeurs, des librairies telles que `OpenMP` ou `Pthread` sont utilisées pour accéder au niveau 3 de parallélisme qui consiste à répartir l'application sur les différents coeurs. Les accélérateurs de type GP-GPU peuvent être programmés à l'aide d'`OpenCL` ou de `CUDA`. Ces librairies sont basées sur le modèle de programmation `fork/join` (voir figure 2.14). Lorsqu'une partie du code doit être exécutée en parallèle, le thread *maître* se dédouble (*fork*) en plusieurs threads pouvant être exécutés indépendamment sur différents coeurs. Une fois les tâches réalisées, les threads sont arrêtés et le programme continue son exécution séquentiellement. Ce modèle de programmation utilisant la même mémoire partagée, le programmeur doit veiller à ce que les différents threads utilisent des données en commun. Un avantage d'`OpenMP` est sa facilité d'utilisation pour exprimer le parallélisme depuis le code source de l'application. Cependant, pour des boucles plus complexes (présence de dépendances entre itérations) il peut être difficile d'atteindre les performances théoriques. L'utilisation d'`OpenMP` nécessite alors une bonne expérience et de bonnes connaissances de la microarchitecture. Pour utiliser `OpenMP`, le code doit être annoté à l'aide de directives préprocesseur `#pragma`. Différentes options permettent de définir le nombre de threads à générer, la façon de partager le travail ou encore définir le mode d'accès aux données (partagé, privé). Le code listé dans l'extrait 2.2 permet de répartir les différentes itérations d'une boucle (indépendantes) entre les threads.

```
1 #pragma omp parallel for
2 for (i=0;i<20;i++)
3     a[i]=b[i] + 42;
```

Extrait 2.2 – Distribution des itérations d'une boucle à l'aide d'OpenMP

9. Graphique adapté de https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

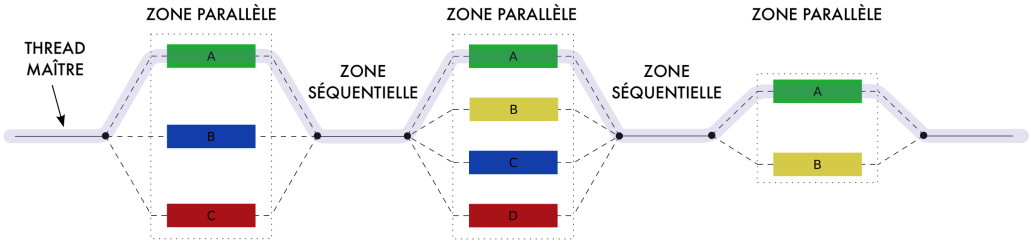


FIGURE 2.14 – Pour accéder au parallélisme des coeurs, OpenMP crée des *threads* indépendants [9].

2.1.4 Performance de la parallélisation

L'utilisation d'un supercalculateur a pour objectif d'accélérer l'exécution d'une application. Cette accélération peut alors permettre d'obtenir des résultats plus rapidement ou bien d'étudier des problèmes plus complexes. L'accélération d'une application, notée $Speedup(P)$, correspond au rapport entre le temps nécessaire pour exécuter l'application sur une ressource unique (noté $T_{sequentiel}$) et le temps nécessaire à l'exécution de la même application lorsque P ressources identiques sont utilisées (noté $T_{parallele}$). L'accélération est dite optimale lorsque le temps de résolution $T_{parallele}(P)$ évolue linéairement avec le nombre P de ressources. On parle alors d'accélération linéaire :

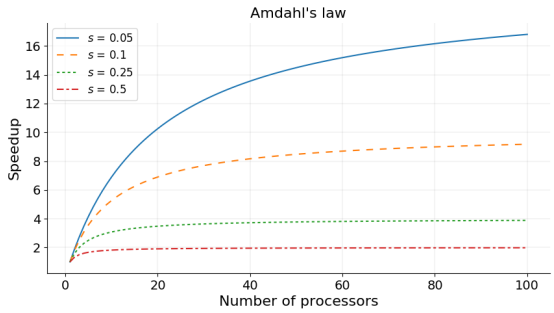
$$Speedup_{lineaire}(P) = \frac{T_{sequentiel}}{T_{parallele}(P)} = P \tag{2.2}$$

Prenons l'exemple d'une application dont le temps nécessaire à l'exécution prend $T_{sequentiel} =$ 2 minutes. Une accélération linéaire avec l'utilisation de $P = 4$ processeurs pendrait alors $T_{parallele} = 30$ secondes. En pratique il est très rare d'obtenir une telle accélération, car certaines parties du code ne peuvent pas être parallélisées (transmission des données en programmation à mémoire distribuée, zone critique en programmation à mémoire partagée, synchronisation). Pour évaluer la capacité d'une plateforme à utiliser efficacement les ressources supplémentaires pour l'exécution d'une application, il est courant de mesurer sa capacité de montée en charge appelée scalabilité. La scalabilité d'une application est un indicateur permettant d'évaluer sa capacité à *passer à l'échelle*, c'est-à-dire d'évaluer l'accélération de son exécution lorsque des ressources de calculs supplémentaires sont allouées. On distingue alors deux scalabilités : la forte et la faible (voir figure 2.15). Les résultats des tests de scalabilité forte et faible permettent de choisir un nombre adapté de ressources à utiliser pour une application.

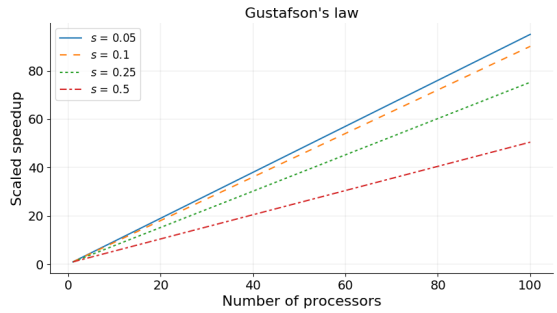
2.1.4.1 Scalabilité forte

En 1967, l'ingénieur Gene Amdahl a étudié et théorisé l'évolution de l'accélération d'une application avec l'ajout de ressources de calculs, créant ainsi la loi éponyme [Amd67]. La taille

10. Source des graphiques - https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling



(a) Loi d'Amdahl : la scalabilité forte utilise un problème de complexité constante.



(b) Loi de Gustafson : la scalabilité faible utilise un problème de complexité croissante.

FIGURE 2.15 – Évolution de la scalabilité forte et faible lorsque P processeurs sont utilisés pour l'exécution d'une application ayant une proportion s de code séquentiel [10].

du problème étant fixe, chaque unité de calcul a donc moins de travail à réaliser. Idéalement, le temps d'exécution doit être réduit d'un facteur $1/P$ lorsque P ressources sont ajoutées. Une application de calcul parallèle n'étant jamais totalement parallélisable, elle possède toujours une proportion de code devant être exécutée séquentiellement. Cette proportion est notée s. L'ajout de ressources de calcul n'est alors bénéfique que pour la proportion de code dit parallélisable et notée $1 - s$. Le temps nécessaire pour l'exécution de l'application est la somme du temps passé dans la zone séquentielle et de celui passé dans la zone parallèle. En utilisant P processeurs, le temps de l'exécution parallèle peut être calculé ainsi :

$$T_{parallele}(P) = s \times T_{sequentiel} + \frac{(1-s) \times T_{sequentiel}}{P} \tag{2.3}$$

L'accélération de l'application peut alors être calculée grâce à l'équation 2.2 donnant lieu à l'équation suivante, appelée loi d'Amdahl :

$$Speedup(P) = \frac{T_{sequentiel}}{s \times T_{sequentiel} + \frac{(1-s) \times T_{sequentiel}}{P}} = \frac{1}{s + \frac{1-s}{P}} \tag{2.4}$$

L'accélération d'une application dépend donc du nombre de ressources de calculs supplémentaires allouées à la résolution de l'application, mais aussi de la proportion de zones exécutables en parallèle. La figure 2.15a montre l'importance de l'influence de la proportion de code séquentiel s sur l'accélération de l'application. Grâce à la loi d'Amdahl, il est possible de donner la limite théorique de l'accélération $Speedup$ d'une application (notée $Speedup_{max}$), lorsque des ressources de calculs supplémentaires sont utilisées pour la résolution d'un problème de taille fixe. L'accélération maximale d'une application utilisant 95% de son temps d'exécution dans une fonction entièrement parallélisable peut être calculée par la limite suivante :

$$Speedup_{max} = \lim_{p \to \infty} \frac{1}{0.05 + \frac{0.95}{P}} = \frac{1}{0.05} = 20 \tag{2.5}$$

Quel que soit le nombre de ressources allouées à l'exécution de cette application, l'application étudiée en exemple ne pourra jamais être accélérée de plus de 20 fois.

2.1.4.2 Scalabilité faible

La loi d'Amdahl est très utile pour montrer la nécessité de développer des applications avec la plus grande portion de code parallélisable. Cependant, la loi suppose une taille de problème constante. Lorsque des ressources de calculs supplémentaires sont disponibles, les applications utilisent généralement des jeux de données plus grands pour profiter de l'espace mémoire supplémentaire. La loi d'Amdahl supposant un jeu de donnée fixe n'est alors pas adaptée. Pour y remédier, Gustafson énonça une nouvelle loi en 1988 [Gus88] permettant de prendre cet aspect en considération. Lorsque la taille du jeu de données augmente, la partie séquentielle du programme représente généralement une plus faible portion, car les données sont traitées dans les zones parallèles. Pour une application ayant une proportion de $1 - s$ pouvant être exécutée en parallèle sur P processeur, l'accélération peut être calculée par la loi de Gustafson :

$$Speedup(P) = s + (1 - s) * P \tag{2.6}$$

La scalabilité faible utilise une taille de problème qui évolue à mesure que des ressources supplémentaires sont allouées. Dans ce cas-là, la taille du problème pour chaque unité de calcul est fixe. Dans l'idéal, le temps d'exécution de l'application doit rester constant lors de l'ajout de ressources de calculs (voir figure 2.15b). Contrairement à la scalabilité forte, l'accélération obtenue par la scalabilité forte n'a pas de limite théorique.

2.2 Évolution de la performance des supercalculateurs

La performance des supercalculateurs a beaucoup évolué depuis leur apparition. Pour comprendre les défis que l'industrie doit relever pour poursuivre ces améliorations, nous nous sommes intéressés à l'évolution des performances des architectures ces 20 dernières années.

2.2.1 Comparer la performance des supercalculateurs

Pour mesurer et comparer la performance des supercalculateurs, il est nécessaire d'avoir une application de référence. L'étalonnage (*benchmarking*) est une pratique courante qui consiste à évaluer plusieurs solutions en leur faisant passer une épreuve commune. Dans le domaine informatique, cela permet de tester différentes architectures matérielles et d'évaluer leurs performances. Un des benchmarks les plus utilisés dans le domaine du HPC est celui développé par Jack Dongara en 1988 : le benchmark HPL (High-Performance Linpack) [DLP03]. Ce benchmark est un code simple qui résout un système d'équations linéaires $Ax = B$, A et B étant deux matrices. Les performances évoluent linéairement avec le nombre de machines utilisées, car très peu de communications sont nécessaires. Le résultat est un nombre d'opérations à virgule

flottante par seconde (FLOPS) que la machine est capable d'exécuter, ce qui rend la comparaison entre supercalculateurs aisée. Ce code permet de publier deux classements bisannuels : le Top500 et le Green500 (figure 2.16).

2.2.1.1 Le Top500

Le TOP500 [12] est un classement mondial qui classe, tous les 6 mois depuis 1993, les 500 supercalculateurs les plus puissants au monde. Ce classement se base sur le nombre maximum d'opérations flottantes qui peuvent être exécutées en une seconde (FLOPS) lors de l'exécution du benchmark HPL. Cette unité a été choisie, car la grande majorité des codes utilisés dans les domaines précédemment cités exécutent des opérations sur des nombres à virgule flottante. Il s'avère donc judicieux de choisir ce dénominateur commun pour comparer les différentes architectures.

Il faut cependant savoir que ce classement ne contient pas toutes les machines. En effet, certains industriels ne préfèrent pas apparaître dans ce classement. Stratégiquement parlant, il peut être intéressant de ne pas publier sa puissance de calcul et nous savons que certaines des infrastructures les plus puissantes n'y figurent pas. Cependant, ce classement permet d'apprécier les tendances que suivent la majorité des architectures.

Au moment de l'écriture de cette section, le dernier classement disponible est celui de novembre 2019 [13]. Les principales caractéristiques des quatre premiers supercalculateurs sont présentées dans le tableau 2.1. Les États-Unis et la Chine possèdent chacun deux entrées et se partagent 70% des performances totales du Top500 (32.3% pour la Chine, 37.1% pour les États-Unis). Concernant le système d'interconnexion, 6 des 10 premières entrées, ainsi que 141 des 500 supercalculateurs, utilisent la technologie Infiniband. L'efficacité des supercalculateurs représente la capacité d'une plateforme à atteindre la performance théorique `Rpeak`. Sur les 500 plateformes répertoriées, 100 d'entre elles ne parviennent pas à atteindre plus de 50% de la performance théorique. Cette information est importante pour comprendre la difficulté qu'ont les applications à utiliser ces plateformes efficacement. Le benchmark `HPL` est un code très simple, avec peu d'accès mémoire. Les applications réelles, plus complexes, ont beaucoup de mal

11. supercomputing.org/
12. www.top500.org
13. Top500 novembre 2019 - www.top500.org/lists/2019/11/

(a) Top500

(b) Green500

FIGURE 2.16 – Les classements du Top500 et du Green500 sont présentés deux fois par an lors de la conférence Supercomputing [11]

Pos.	Nom	Country	Nb. coeurs	Coeurs accélérateur	Alimentation (MW)	Accélérateur	Rpeak	Rmax	Éfficacité
1	Summit	United States	2414592	2211840	10	NVIDIA GPU	200	148	0.74
2	Sierra	United States	1572480	1382400	7	NVIDIA GPU	125	94	0.75
3	Sunway TaihuLight	China	10649600	0	15	None	125	93	0.74
4	Tianhe-2A	China	4981760	4554752	18	Matrix-2000	100	61	0.61

Tableau 2.1 – Classement du Top500 de novembre 2019. Les puissances Rpeak (puissance théorique) et Rmax (puissance mesurée par HPL) sont données en pétaFLOPS (10^{15} FLOPS). L'efficacité est le rapport entre Rmax et Rpeak.

Position	TOP500	Processeur	Accélérateur	Rmax	Rpeak	Efficacité	Efficacité énergétique
1	159	Fujitsu A64FX	Aucun	1.99	2.35	0.84	16.876
2	420	Xeon D-1571	PEZY-SC2 700Mhz	1.30	1.79	0.72	16.256
3	24	IBM POWER9	Volta GV100	8.04	11.12	0.72	15.771
4	373	IBM POWER9	Tesla V100 SXM2	1.46	1.73	0.84	15.574

Tableau 2.2 – Classement du Green500 de novembre 2019 selon l'efficacité énergétique (en gigaFLOPS/Watts). Les puissances Rpeak (puissance théorique) et Rmax (puissance mesurée par HPL) sont données en pétaFLOPS (10^{15} FLOPS).

à atteindre la performance théorique. Depuis quelques années, le classement du Top500 donne la performance d'un second benchmark (HPCG, voir section 2.4.1.2). Pour ce benchmark, plus fidèle aux comportements d'applications réelles, aucune des plateformes n'atteint une efficacité supérieure à 4%. Pour avoir une meilleure vision de l'efficacité des plateformes, le classement du Green500 peut être consulté.

2.2.1.2 Le Green500

Un second classement a vu le jour en 2007 appelé le Green500 [14]. Il classe les supercalculateurs selon un critère d'efficacité énergétique. Cette efficacité mesure le nombre d'opérations sur un nombre flottant réalisé pour 1 watt d'énergie fournie au supercalculateur.

En effet, ces vingt dernières années, les évolutions technologiques et l'augmentation de la taille des supercalculateurs n'avaient pas de limite physique. Cette course à la performance n'était dictée que par les budgets disponibles pour leur construction. Ainsi, l'émergence de plateformes consommant de grandes quantités d'énergie et très peu efficaces a été constatée (voir le classement du Top500).

En novembre 2019 [15], les 34 premiers supercalculateurs utilisent tous des accélérateurs. La majorité étant des GPU Nvidia Tesla. Il est intéressant de noter que la deuxième place du classement est tenue par une plateforme basée sur le processeur PEZY-SC2 conçu par l'entreprise PEZY et fabriqué par TSMC. Seul le premier du classement n'utilise pas d'accélérateurs (voir tableau 2.2). Il utilise un processeur ARM basse fréquence pouvant exécuter des instructions vectorielles SVE (Scalable Vector Extensions) sur 512 bits.

14. Green500 - www.top500.org/green500/
15. www.top500.org/green500/lists/2019/11/

2.2.2 Évolution des performances des supercalculateurs

Le classement du Top500 a de nombreux avantages, dont celui de promouvoir le HPC au grand public. Cependant, un effet de bord de ce classement est de motiver les constructeurs pour développer des architectures ayant de bonnes performances lors de l'exécution du benchmark HPL. Or, ce code n'est pas représentatif des applications réelles et nous pouvons nous demander si son utilisation pour l'établissement du Top500 n'a pas été contre-productive. Bien que certains des supercalculateurs les plus puissants n'y figurent pas, l'évolution du Top500 permet d'obtenir un aperçu des évolutions de performances des architectures.

Pour avoir une vision globale de son évolution, nous étudions la performance cumulée des 500 supercalculateurs depuis le premier classement (voir figure 2.17). Nous distinguons deux phases : avant et après 2012. Les performances du Top500 ont évolué d'un facteur 1000 tous les 11 ans, conformément aux performances prédites par la loi de Moore. À partir de 2013, il faut attendre en moyenne 20 ans pour voir la performance des supercalculateurs augmenter dans la même proportion. Dans cette section, nous discutons des différents facteurs qui ont permis l'évolution constante des performances des architectures pendant près de 20 ans et étudions les freins qui empêchent de poursuivre ce rythme. La majorité des améliorations évoquées sont présentées dans l'Annexe A.

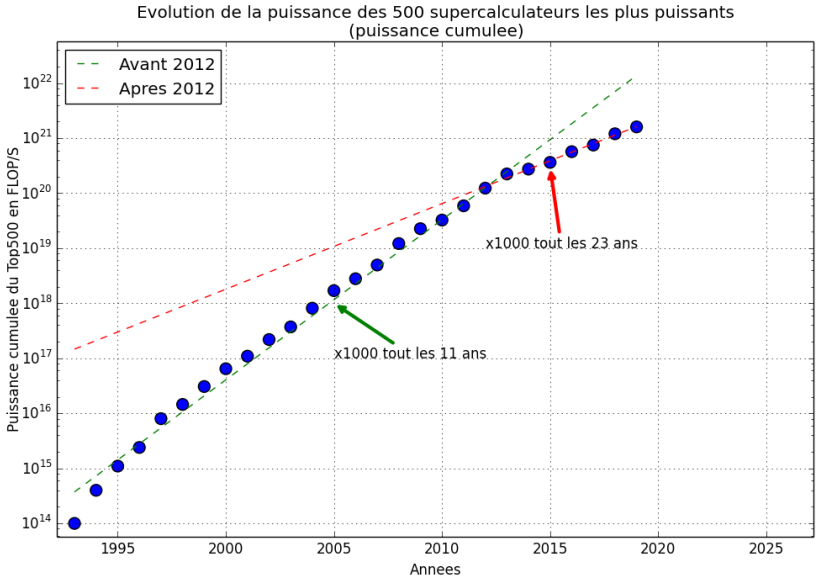
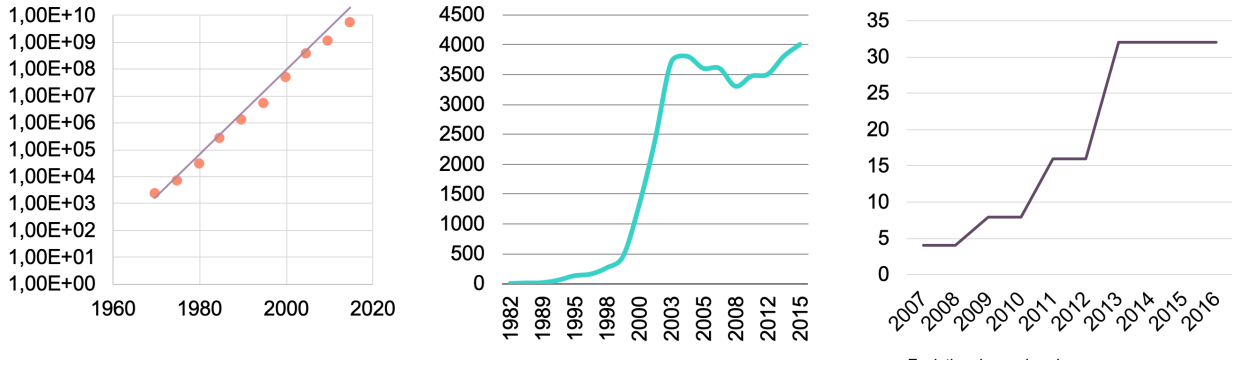


FIGURE 2.17 – Évolution de la performance cumulée des 500 supercalculateurs les plus puissants au monde. La pente de l'évolution diminue à partir de 2012.

2.2.2.1 Avant 2012

Les supercalculateurs ont pu bénéficier de nouvelles technologiques ainsi que de nombreuses innovations techniques.



(a) Évolution du nombre de transistors (Annexe A.2.3.2)

(b) Évolution de la fréquence (Annexe A.2.3.1)

(c) Évolution du nombre de coeurs (Annexe A.2.4.3)

FIGURE 2.18 – Évolutions technologiques principales des processeurs. Ces différentes évolutions sont présentées dans l'Annexe A.

Les processeurs. Dans l'Annexe A, nous présentons les différentes évolutions technologiques dont ont pu bénéficier les processeurs. La figure 2.18 résume les trois principales évolutions. Grâce à l'affinement des procédés de gravure, le nombre de transistors a évolué exponentiellement (voir figure 2.18a). En miniaturisant les transistors et en utilisant des systèmes de refroidissement plus efficaces, la fréquence des processeurs a pu être augmentée de plusieurs facteurs (voir figure 2.18b). Lorsque la fréquence n'a plus pu être augmentée, les architectures parallèles ont été développées, donnant naissance aux processeurs multicoeurs (voir figure 2.18c). La microarchitecture elle-même a reçu de nombreuses améliorations : l'utilisation de pipeline (voir Annexe A.2.4.1) pouvant être superscalaire (voir Annexe A.2.4.2), ainsi que le développement d'unités de calculs pouvant exécuter des instructions vectorielles (voir Annexe A.2.2.2).

Les mémoires. La performance des processeurs évoluant, celle des mémoires a aussi dû être améliorée pour fournir les données nécessaires aux traitements plus rapidement. Les principales améliorations sont dues à l'utilisation de nouvelles technologies mémoires (voir section A.3.1), l'augmentation du nombre de canaux reliant la mémoire au processeur ou encore l'implémentation d'une hiérarchie de mémoire (voir section A.3.2). Celle-ci permettant aux applications de profiter du principe de localité (voir section A.3.2.3).

2.2.2.2 Après 2012

En étudiant l'évolution de la puissance des supercalculateurs (voir figure 2.17), nous remarquons un ralentissement à partir des années 2010-2012. Ce ralentissement n'est pas dû à un seul facteur, mais à un ensemble de contraintes. En effet, les leviers et évolutions technologiques qui permettaient de tenir cette cadence ne sont plus disponibles aujourd'hui ou sont en fin de course (voir figure 2.18). Si certaines lois ont assuré une évolution continue de la performance des processeurs pendant plusieurs dizaines d'années, une partie du ralentissement de l'évolution

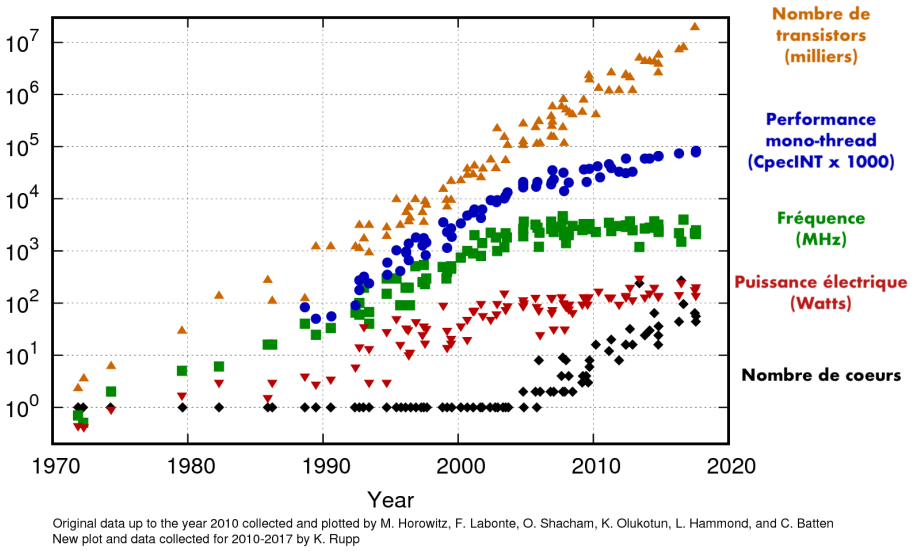


FIGURE 2.19 – Évolution des principales caractéristiques des processeurs (données originales tirées de [Rup15] [16]).

des performances du Top500 peut être expliquée par leur *essoufflement* [Fra15].

Fin de validité de la loi de Moore. La loi de Moore [Moo65] prévoyait que les architectures pourraient doubler leur nombre de transistors, tous les deux ans [Moo75], à coût constant (voir section A.1.6.1). Malheureusement, les fondeurs ne parviennent plus à suivre le rythme dicté par la loi énoncée par Gordon Moore, pour des raisons principalement techniques (gravure), de coût [Bro17] et de limite physique. La miniaturisation continue des transistors, dont la taille actuelle est de quelques nanomètres, rend la circulation des courants électriques instable. La bonne circulation des signaux électriques dans les circuits ne pouvant plus être garantie, il n'est alors plus possible de réduire leur taille. À partir de 2013, l'évolution des performances du Top500 passe pour la première fois sous les performances prévues par la loi de Moore (figure 2.20).

Fin de la validé de la loi de Dennard. Dans l'Annexe A.2.3.2, nous discutons de l'augmentation de la fréquence des processeurs et introduisons la loi de Dennard [Den+74]. Ces équations ont prédit l'augmentation de la fréquence des processeurs de 40% tous les 2 ans pendant plus de 30 ans. À cause des courants de fuite [WM95] augmentant exponentiellement avec la finesse de gravure des transistors, la consommation électrique des processeurs a elle aussi augmenté. Dans la section A.2.3.2, nous expliquons comment la réduction de la taille des transistors et l'augmentation de la fréquence des processeurs augmentent la consommation électrique des circuits. L'énergie utilisée par le processeur étant dissipée sous forme de chaleur, il est devenu très difficile de refroidir ces architectures. De plus, l'énergie nécessaire pour le refroidissement a un

16. Les données sont accessibles sur le dépôt GitHub de l'auteur : https://github.com/karlrupp/microprocessor-trend-data

impact sur la consommation. Cette limitation physique empêchant d'augmenter la puissance électrique des processeurs est appelée *power wall* [Kur01].

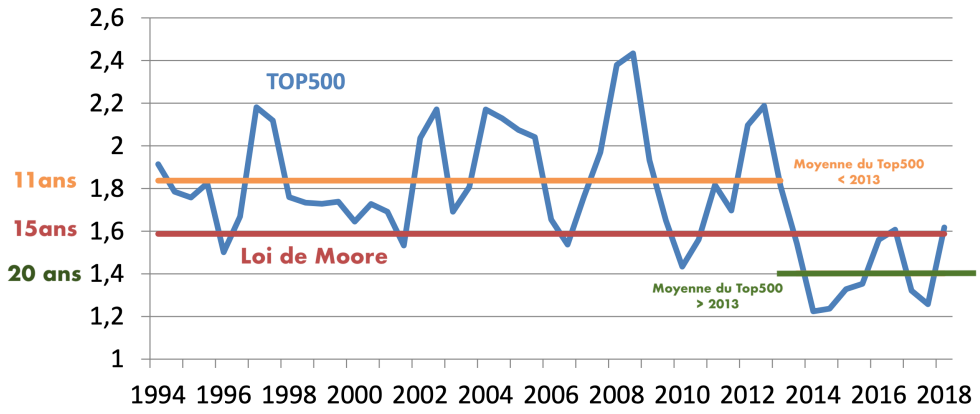


FIGURE 2.20 – Facteur d'évolution annuelle des performances du Top500 [17]. Jusqu'à 2013 la performance moyenne du Top500 évoluait d'un facteur 1.8, supérieur à l'évolution prédite par la loi de Moore (facteur 1.6).

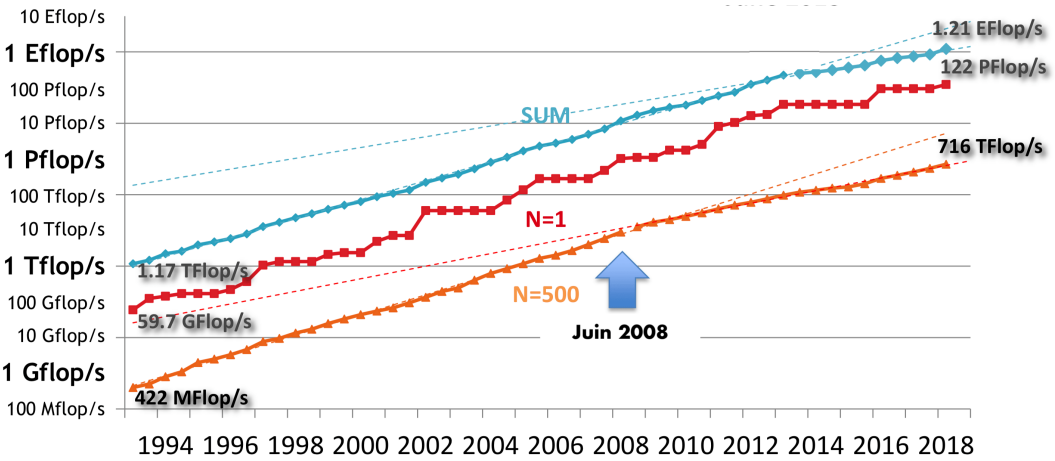


FIGURE 2.21 – Évolution des performances du Top500 en FLOPS grâce à l'aide du benchmark HPL. Le graphe présente la puissance cumulée des 500 ordinateurs (en bleu), celle du premier (en rouge) et celle du dernier (en orange) [18].

Budget. Le prix des supercalculateurs est un frein à la construction de centres toujours plus puissants. Les calculateurs les moins puissants du classement sont généralement ceux dont le budget est le plus faible. Grâce à l'analyse du Top500, nous pouvons estimer que l'économie est un des freins participant au ralentissement de l'évolution des performances des architectures

18. Graphique inspiré de la présentation du Top500 lors de la conférence ISC18 - *Highlights of the 51st TOP500 List*

(voir figure 2.21). Le décrochage des performances du Top500, autour de 2012, étudié dans la section précédente intervient plus de 4 ans après le décrochage du dernier du classement (en 2008). De plus, on peut étudier la répartition de la performance du Top500 entre les supercalculateurs. En 2004, il fallait agréger les 80 premiers supercalculateurs du classement pour obtenir la moitié de la performance cumulée du Top500. En 2019, il faut seulement cumuler la puissance des 28 premiers. Ceci montre que les plateformes en haut du classement ont tendance à augmenter leur performance plus vite que celles du bas.

Déséquilibre des microarchitectures. Si l'utilisation de processeurs multicoeurs a permis de continuer d'augmenter la performance des processeurs malgré la faible évolution des fréquences, l'ajout de coeur a lui aussi atteint ses limites. En effet, comme démontré dans la section 2.1.4, l'augmentation des niveaux de parallélisme atteint elle aussi ses limites lorsque les applications contiennent des zones de codes séquentiels. Même des applications comme le benchmark `HPL` sont impactés par **la loi d'Amdahl**, et l'ajout de coeurs s'est révélé de moins en moins efficace. On remarque sur la figure 2.19 que le nombre de coeurs a peu évolué ces dix dernières années. Dans l'Annexe A.3.1.2, nous discutons de la disparité des évolutions technologiques des processeurs d'une part et des mémoires d'autre part. Cet écart a évolué au fil des années et il est aujourd'hui appelé *mur de la mémoire* [Roj97] (*memory wall* ou *memory gap*). Les premiers processeurs étaient limités par la puissance de calcul, aujourd'hui il est très rare que la performance des applications de calcul haute performance soit limitée par celle des unité de calcul à virgule flottante (FPU). L'ajout supplémentaire de coeurs est donc moins bénéfique, car le ratio de bande passante disponible par coeur diminue. En étudiant l'évolution des performances des différentes parties du système, nous observons alors de réelles différences :

- La performance calculatoire des processeurs (le nombre d'opérations flottantes réalisables par cycle) a **augmenté de 50%** en moyenne par an.
- La bande passante entre le processeur et la mémoire a augmenté de 23% par an ;
- La latence des requêtes mémoires a **diminué de 4%** par an ;
- La débit réseau a **augmenté de 20%** par an.

La figure 2.22 montre le déséquilibre entre ces deux parties fondamentales des architectures Von Neumann. Ce déséquilibre empêche aujourd'hui le système mémoire de transférer les données suffisamment rapidement pour que la totalité des unités de calculs soit constamment active. Pourtant, 50% des broches d'un processeur récent sont allouées au système mémoire. Ainsi, bien que les performances de calculs des processeurs s'améliorent, les applications ne peuvent pas en bénéficier. La plupart des supercalculateurs atteignent rarement une efficacité de 80% sur une application simple comme Linpack [DLP03]. Pour des applications réelles, cette efficacité est encore plus faible, parfois inférieure à 10% [Oli+05].

Le déséquilibre des microarchitectures se fait d'autant plus ressentir lorsque des centaines de serveurs sont réunis. La figure 2.23 montre l'évolution de la puissance des serveurs et du débit mémoire des 10 premiers supercalculateurs du Top500. Grâce aux évolutions des processeurs (voir section 2.2.2.1) et l'utilisation des GPUs, la puissance des serveurs a été multipliée en moyenne par 65 entre 2010 et 2018 (courbe bleue). Pendant la même période, le débit des

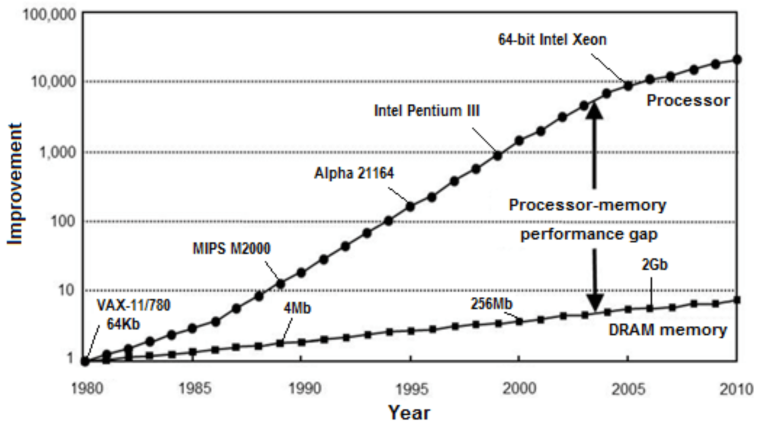


FIGURE 2.22 – Progression de la performance des processeurs et des mémoires. La performance de plusieurs générations de processeurs a été mesurée à l'aide du benchmark SPECint [ECT17]. La performance mémoire est représentée par la latence des accès mémoire (CAS et RAS) des mémoires DRAM.

communications interserveur n'a lui augmenté que d'un facteur 4.8 (courbe rouge). Ainsi, le ratio entre la puissance de calcul des serveurs et le débit de communication (byte par FLOP) n'a fait que diminuer. Entre 2017 et 2018, ce ratio a diminué d'un facteur 8 pour les 2 premiers supercalculateurs du Top500 (respectivement Sunway Taihulight et Summit). Pour des applications nécessitant de grandes communications interserveur, ce ratio très faible les empêche d'atteindre plus d'une fraction de la performance disponible.

2.2.3 Le futur du HPC

> Peu importe la puissance qu'atteindront les processeurs, le logiciel trouvera toujours une façon d'utiliser cette puissance. Construisez un processeur 10 fois plus rapide, et la partie logicielle trouvera toujours 10 fois plus à faire (ou le fera 10 fois moins efficacement) [SL05].

2.2.3.1 Situation du HPC en 2020

La performance des supercalculateurs et des technologies de l'information n'a fait qu'augmenter au cours de ces 30 dernières années. Une montre connectée récente apporte une puissance de calcul deux fois supérieure à celle du supercalculateur Cray-2, le plus puissant des supercalculateurs de 1985. Un GPU moderne tel que le GPU Nvidia V100 délivrant une puissante de 7.5 téraFLOPS (10^{12} FLOPS) serait classé à la 30e place du Top500 de 1994.

L'évolution des performances des matériels informatiques a largement transformé le mode de vie de nos sociétés. Les produits du HPC sont présents dans nos vies quotidiennes que ce soit lorsque nous nous déplaçons grâce à l'essence extraite à l'aide de la modélisation des fonds

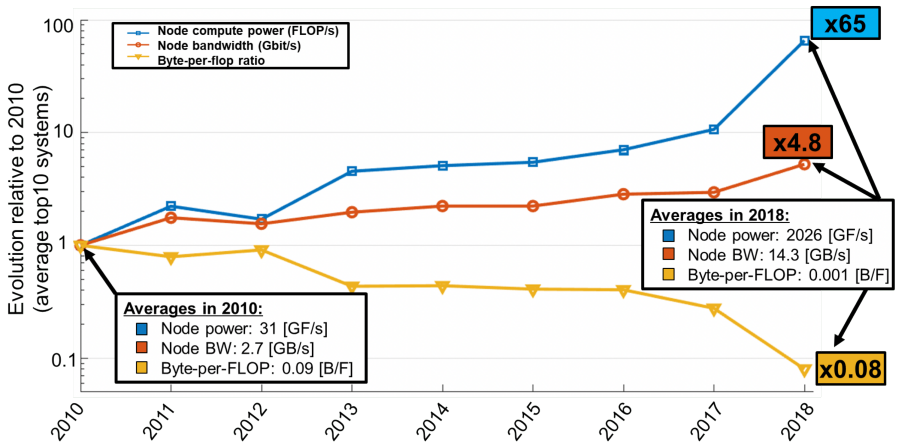


FIGURE 2.23 – Évolution des 10 premiers supercalculateurs du Top500 de 2010 à 2018 : la puissance de calcul des serveurs (FLOP/s) en bleu, le débit de communication entre les serveurs (Gbit/s) en orange, ainsi que le ratio entre ces deux caractéristiques (byte par FLOP) en jaune. (graphique tiré de la conférence IPDPS 2018 [Ber18])

marins ou lorsque nous consultons la météo. Le HPC a un réel impacte sur nos vies, les rendant plus sécurisées en prévoyant précisément des catastrophes naturelles.

Suite à la forte évolution de la performance des supercalculateurs, nous pouvons nous demander s'il est nécessaire de continuer à construire des infrastructures toujours plus puissantes. La variété et la complexité des problèmes qui peuvent être traités dépendent directement de la puissance de calcul disponible. L'apprentissage profond (*deep learning*) en est un bon exemple. Dès 1989, Yann LeCun améliorait déjà la rétropropagation [THW12] et utilisait les réseaux neuronaux convolutionnels [LeC89]. Mais à cette époque les GPGPU n'étaient pas disponibles et les CPU étaient loin d'être assez puissants pour exécuter ces algorithmes, même sur les données limitées disponibles à cette époque. Dans les années 2010, les GPU sont devenus suffisamment puissants pour permettre l'exécution de ces algorithmes.

2.2.3.2 Nécessité de construire des plateformes plus puissantes

> La taille des données double tous les deux ans, et d'ici 2020, les données que nous générons et copions annuellement atteindront 44 Zettabytes (10^{21} bytes) [Zha+17].

La mise au point de plateformes plus puissantes va permettre aux entreprises d'être plus compétitives et aux équipes de recherches de réaliser de nouvelles découvertes. Un grand nombre de domaines scientifiques vont pouvoir profiter de telles puissances de calcul : découverte de nouveaux matériaux, simulations de réactions chimiques ou encore pour analyser les résultats d'expériences lourdes comme celles réalisées au Grand collisionneur de hadrons [Blo+17]. La recherche pour le climat va aussi bénéficier de telles architectures et permettre de comprendre les dérèglements climatiques, anticiper la hausse des océans et élaborer de meilleurs modèles. Les

simulations permettront également d'améliorer l'efficacité et la sécurité des réacteurs nucléaires [SZS07]. En astrophysique, elles permettront d'étudier des phénomènes encore incompris comme la formation des trous noirs [Ber+13]. L'accès à des infrastructures plus puissantes permettra de poursuivre les avancées réalisées dans le domaine de la physique quantique. Ces découvertes pourront permettre de construire d'autres plateformes de calculs appelées ordinateurs quantiques.

Aujourd'hui, de nombreuses applications sont prêtes, mais ne peuvent pas être exécutées en un temps raisonnable avec les moyens de calculs disponibles. Actuellement il est possible de simuler des réactions chimiques de quelques nanosecondes et sur de petits volumes. Avec la construction de supercalculateurs plus puissants, il sera possible de réaliser des simulations plus longues et plus précises. Nous discutons dans cette section de trois facteurs importants qui motivent la nécessité de poursuivre le développement de plateformes plus performantes que celles actuellement en notre possession : l'explosion du volume de données à traiter, la complexification des analyses et le besoin d'obtenir des résultats rapidement.

Tsunami de données. Suite aux évolutions des technologies des semi-conducteurs, nous assistons depuis le début des années 2010 à l'explosion des quantités de données générées (voir figure 2.24). Grâce aux nanotechnologies, il est possible de produire des capteurs à très faible coût. La consommation électrique de ces matériels étant faible, ces capteurs sont installés partout. L'Internet des Objets (IOT) interconnecte ces milliers de capteurs pour générer de grands volumes de données (*big data*). Les évolutions récentes de la domotique ont permis d'installer des capteurs dans toute la maison : que ce soit pour gérer les lumières, le chauffage, l'ouverture des volets ou même l'allumage de la cafetière. En 2018, Ericsson comptait plus de 5 milliards d'abonnés au téléphone dans le monde [19]. Ces téléphones peuvent aujourd'hui suivre nos moindres faits et gestes, générant de grandes quantités de données. Celles-ci peuvent ensuite être utilisées par différentes applications pour étudier les déplacements des habitants d'une ville ou proposer des annonces ciblées. Les villes installent des milliers de capteurs permettant le développement de nombreuses applications comme le suivi du trafic routier [BMP16], l'optimisation des transports en commun, ou encore le suivi de la qualité du recyclage des déchets dans un quartier [Reb18]. Ce grand volume de données produit aujourd'hui et qui va s'intensifier dans les années futures est le principal défi des systèmes d'informations.

Complexité des calculs. Aujourd'hui, la valeur ne provient plus de la capacité à produire ces données, mais de la capacité d'en extraire une valeur avec des algorithmes de *big data* et d'apprentissage machine. Les algorithmes d'intelligence artificielle nécessitent d'être entraînés sur des jeux de données pour pouvoir ensuite prendre les décisions adaptées. En 2018, OpenAI a constaté que la puissance de calcul utilisée pour entraîner les plus grands modèles d'IA avait doublé tous les 3,4 mois depuis 2012 (voir figure 2.25). Dans le domaine de la santé, de telles infrastructures pourront permettre d'extraire et analyser toutes les informations de millions de patients atteints de maladies graves. Il sera alors possible de mieux comprendre les symptômes

19. Source : https://www.usinenouvelle.com/article/5-4-milliards-d-abonnes-au-telephone

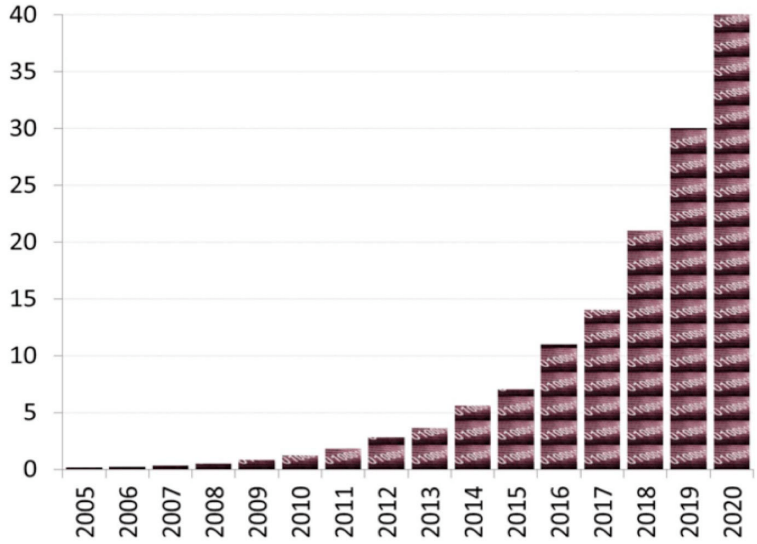


FIGURE 2.24 – Le volume de données généré chaque année devrait atteindre 40 Zettabytes en 2020 [Sim16]

menant au développement d'un cancer ou d'une maladie cardiaque. Pour obtenir des résultats dans des temps raisonnables, ces applications nécessitent d'avoir accès à des plateformes plus puissantes de plusieurs facteurs que celles existant actuellement. Avec l'analyse des données obtenues grâce aux montres connectées, il sera alors possible d'anticiper les maladies cardiaques.

Temps de traitement. La troisième raison qui motive le développement de nouvelles plateformes plus puissantes est de réduire le temps d'exécution des applications. Une fois la donnée produite, sa valorisation diminue dans le temps. La météo est un très bon exemple d'application ne pouvant pas excéder un certain temps de traitement si les prédictions veulent être données suffisamment rapidement. Par exemple, pour permettre aux voitures connectées d'identifier un danger, l'action correspondante doit être prise le plus rapidement possible. La possibilité d'envoyer les données à traiter à un supercalculateur n'est alors pas envisageable. L'accès à des plateformes plus puissantes est alors nécessaire pour traiter cette quantité de données dans des temps raisonnables. En plus d'améliorer les performances des supercalculateurs, l'architecture complète des plateformes doit être repensée. Pour traiter efficacement le volume de données générées, plusieurs travaux montrent qu'il est indispensable de repenser notre façon de stocker, déplacer et analyser ces données [Sal18 ; CML14 ; NCC17 ; GR15].

2.2.3.3 Exascale

Le terme exascale est utilisé pour définir la prochaine génération de supercalculateurs capable d'atteindre une performance de 10^{18} FLOPS (un exaFLOPS). Cette performance devra être mesurée à l'aide du benchmark High-Performance Linpack (HPL [DLP03]). En effet, certains supercalculateurs sont déjà capables d'atteindre de telles performances sur des applications

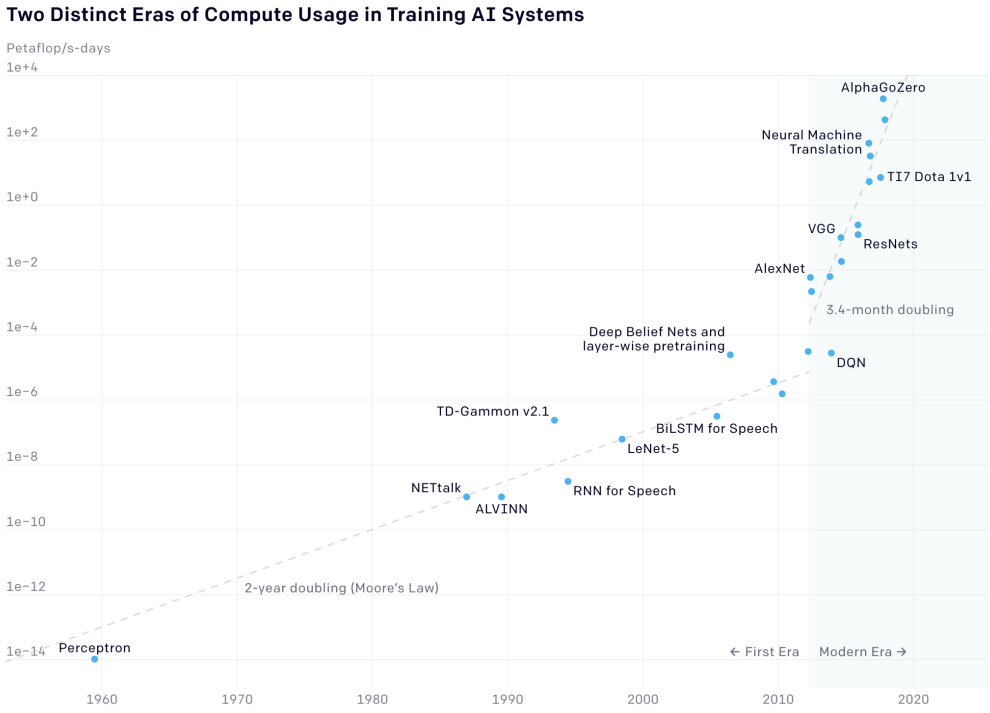


FIGURE 2.25 – Quantité totale de calcul (en pétaFLOPS (10^{15} FLOPS)) nécessaire chaque jour pour entraîner les différents réseaux de neurones [AH18].

plus simples. En 2018 le supercalculateur Summit était capable d'exécuter $1.2 * 10^{18}$ opérations par seconde. L'application utilisée était une application d'apprentissage en profondeur utilisée pour étudier les déchets nucléaires. Cette application n'utilise que des opérations en simple précision. Le même supercalculateur a pu atteindre une performance de 3.3×10^{18} opérations sur une application d'analyse de données, elle aussi en simple précision [Ber18]. [20].

Dans la littérature, il est courant d'utiliser le terme *exascale* pour désigner cette nouvelle génération de plateformes 10 fois plus puissantes que les supercalculateurs les plus puissants actuels, et 30 fois plus puissants que la moyenne du Top500 de novembre 2019 (33 pétaFLOPS). Le rapport commandé par le département de l'énergie américain en 2014 [Luc+14] souligne que ce terme ne fait pas référence à la puissance obtenue par le benchmark HPL, mais bien à une plateforme avec 1000 fois plus de capacité que les supercalculateurs existant alors. Dans ces travaux de thèse, nous suivons la même démarche et utilisons le terme exascale pour désigner la plateforme qui permettra d'obtenir les performances souhaitées. La plateforme ne concerne pas seulement le supercalculateur, mais toute la chaîne de traitement de données, depuis sa création jusqu'à son traitement.

2.2.4 Défis à relever pour l'élaboration d'une plateforme exascale

Jusqu'aujourd'hui, quand une piste d'évolution venait à s'épuiser, comme l'évolution de la fréquence des processeurs, d'autres améliorations venaient alors y pallier (augmentation du nombre de coeurs). Dans la section 2.2.1.1, nous avons discuté des principaux freins ayant ralenti l'évolution des performances des supercalculateurs ces 8 dernières années. Depuis plusieurs années, de nombreuses études [SDM10 ; Kog+08 ; Ber+11] prédisent les principaux challenges à relever pour la construction d'une plateforme exascale. La plus citée d'entre elles est l'étude menée par le département de l'énergie américain [Luc+14] qui décrit les dix principaux challenges à relever. Ce rapport met en lumière les nombreuses difficultés que l'industrie rencontre et montre que celles-ci touchent toutes les parties des centres de données (*data center)*) : l'énergie, les technologies mémoires et d'interconnexion, la programmation, la gestion ou encore la capacité des applications à utiliser la totalité de la puissance disponible. Il est aussi indiqué que pour construire une telle plateforme dans des coûts financiers et des consommations électriques raisonnables, les dernières avancées technologiques des domaines cités précédemment doivent être intégrées. Le travail de la thèse a en partie été motivé par ces challenges présentés dans les études citées ci-dessus. Dans cette section nous présentons en détail les six principales difficultés que nous avons identifiées.

2.2.4.1 Les coûts

Dans la section 2.2.2.2 nous avons expliqué comment la fin de la loi de Moore avait impacté l'évolution des performances des supercalculateurs. Bien que souvent oubliée, il est important de rappeler que cette loi est avant tout une loi économique. Grâce à l'évolution des procédés de

20. https://insidehpc.com/2019/11/deep-learning-on-summit-supercomputer-powers-insights

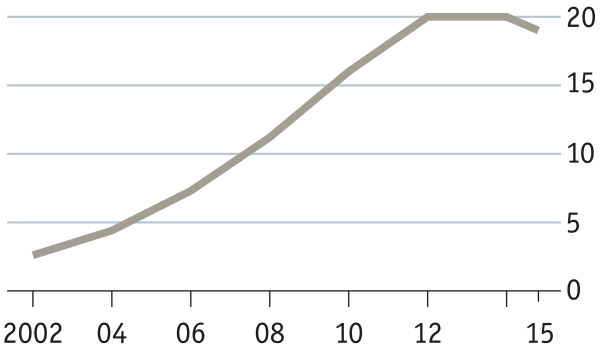


FIGURE 2.26 – Nombre de transistors achetés par dollar dépensé [21].

gravure, le nombre de transistors a pu doubler tous les deux ans pendant plus de trente ans alors que le coût des investissements dans les fonderies n'augmentait que de 30% sur la même période. Cette évolution des prix des fonderies porte le nom de *seconde loi de Moore* ou loi de Rock. À cause des nombreux investissements nécessaires pour développer les nouvelles générations de fonderies nous remarquons que le nombre de transistors achetable pour un budget donné n'évolue plus (voir figure 2.26).

Le prix d'un supercalculateur n'est pas seulement constitué de l'addition du prix des processeurs et des mémoires. Elle prend en compte le Coût Globale de Possession (TCO) (voir section 4.5). Cette valeur prend en compte le cumul des coûts du centre de données durant la totalité de son cycle de vie. Il tient compte du prix du bâtiment dans lequel se trouve le supercalculateur, de sa consommation électrique, mais aussi de l'opérationnel (personnel nécessaire pour sa gestion). Tous ces facteurs doivent être pris en compte, le prix de l'électricité tenant une grande part dans l'équation. Pour les entreprises, le facteur économique est souvent le premier critère de décision. Les budgets alloués au développement et à l'utilisation d'une nouvelle plateforme sont décidés en amont et il est rare que des budgets supplémentaires soient alloués, même pour une plateforme plus performante. Lorsque HPE répond à des appels d'offres pour la construction d'un nouveau supercalculateur, les métriques utilisées sont les $FLOPS/\$$ ou $GB/s/\$$. Cette forte pression économique est une contrainte et il est difficile de proposer des sauts technologiques qui nécessitent de lourds investissements en amont sans connaître les retombées à l'avance. Lorsqu'une entreprise investit dans une solution, elle veut pouvoir quantifier à l'avance les bénéfices qu'elle peut en tirer. Lors de l'analyse du Top500, nous avons remarqué que la vitesse de l'accroissement des performances des supercalculateurs des entités avec les plus petits budgets avait ralenti 4 ans (2008) avant le reste du Top500 (2012).

2.2.4.2 La consommation énergétique

L'investissement dans un supercalculateur ne concerne pas seulement son acquisition. Un budget conséquent doit être alloué à l'alimentation du matériel, mais aussi du système de refroidissement. L'évolution de la consommation des processeurs a nécessité le développement

21. Source : Intel ; rapport de presse ; Bob Colwell ; Linley Group ; The Economist ; IB Consulting

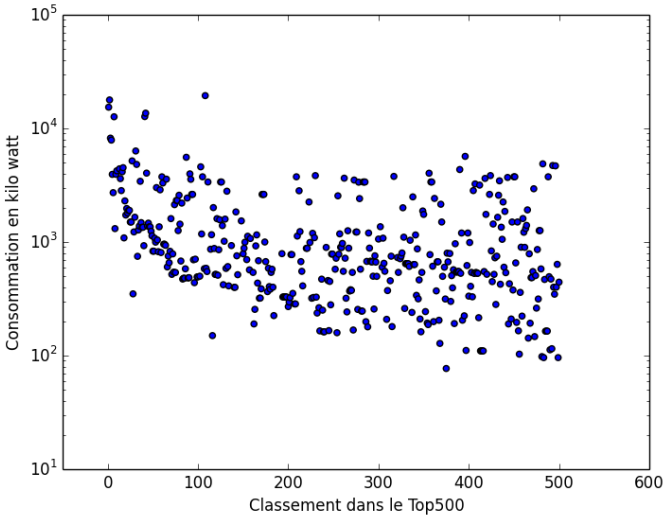


FIGURE 2.27 – Consommation électrique des 500 supercalculateurs les plus puissants (2018).

de systèmes de refroidissement ultra-optimisés. Toute l'énergie étant transformée en chaleur, des techniques telles que le refroidissement à l'eau et l'immersion dans des bains d'huile sont régulièrement utilisées. Si on regarde la consommation électrique des clusters du Top500, on constate qu'en moyenne ils consomment 1,4 mégawatt et que les 5 qui consomment le plus sont au-delà de 12 mégawatts (voir graphique 2.27). En supposant que le prix de l'électricité coûte 1 dollar par watt par année, l'alimentation de ces architectures coûte alors annuellement des millions de dollars (20 millions de dollars pour le premier).

En plus du coût financier, il est important de considérer l'impact écologique de la consommation électrique de ces plateformes. Une étude a été menée pour évaluer l'évolution de l'entraînement des principaux réseaux de neurones (AlexNet, AlphaGo, GoogleNet...) [SGM19]. La quantité de calculs nécessaire pour l'entraînement d'un réseau a été multipliée par 300 000 en 7 ans. L'étude rapporte l'empreinte carbone de l'entraînement de ces réseaux. Le CO_2 émis lors de l'entraînement d'un réseau équivaut au CO_2 émis par un passager réalisant 300 allers-retours en avion entre New York et San Francisco ou encore à l'émission de cinq voitures américaines durant toute leur durée de vie.

Le Département de l'énergie des États-Unis (DoE) a publié son rapport de propositions PathForward, et a établi qu'un supercalculateur exaflopique devra consommer au maximum entre 20 et 30 mégawatts [Ang16]. Actuellement, les quatre premiers supercalculateurs consomment entre 7 et 18 MW. La construction d'une plateforme exascale implique donc de réaliser 30 fois plus d'opérations avec une enveloppe énergétique seulement doublée. L'énergie est une contrainte majeure dans la construction d'une plateforme Exascale. Pour nous en rendre compte, nous étudions le dernier classement du Top500 paru en novembre 2019. Pour obtenir une puissance cumulée d'un exaFLOPS, il faut additionner la puissance des 105 premiers supercalculateurs du Top500. Une telle plateforme nécessiterait alors une alimentation de plusieurs

centaines de mégawatts. Le même exercice peut être réalisé avec le classement du Green500 (voir tableau 2.2), qui classe les architectures les plus efficaces du Top500. Pour obtenir une puissance cumulée d'un exaFLOPS, il faudrait regrouper les 205 premiers supercalculateurs pour obtenir une plateforme consommant 279 MW. Ces rapides calculs permettent de montrer l'étendue vertigineuse des améliorations qui sont nécessaires pour la construction d'une plateforme qui respecterait les critères de consommation [Ang16].

Consommation de la microarchitecture. Ajouter des serveurs pour la construction d'un supercalculateur plus puissant nécessite une plus grande puissance électrique. Cette technique est restée viable pendant plus de vingt ans, alors que les processeurs ne consommaient que quelques watts. Aujourd'hui, les processeurs utilisés ont une enveloppe thermique (TDP) dépassant la centaine de watts : 190 watts pour le processeur IBM Power9 22 coeurs (utilisé dans le supercalculateur Summit), 150 watts pour le processeur Intel Skylake 6148 20 coeurs (utilisé dans le 8e supercalculateur du classement du Top500 2019). Les plateformes les plus puissantes associent à un processeur un ou plusieurs accélérateurs (principalement des GPU). Chaque noeud du supercalculateur Summit possède ainsi 2 processeurs et 6 GPU NVidia V100 ayant chacun un TDP de 250 watts. À de telles consommations, il n'est plus viable d'ajouter des noeuds de calculs indéfiniment, que ce soit pour une raison de coût ou bien de faisabilité. En effet, les sites où sont construits les centres de données ont des lignes électriques déjà existantes et qui ne peuvent pas supporter de plus grandes puissances. Une partie des utilisateurs sont donc limités par cette enveloppe énergétique et doivent l'utiliser le plus efficacement pour la transformer en puissance de calcul. La consommation électrique des architectures varie fortement en fonction des opérations réalisées (voir figure 2.28). Que ce soit pour des architectures 32 bits [Hor14] ou 64 bits [Lel+14] la consommation électrique des opérations élémentaires de la microarchitecture n'a que très peu évolué ces dernières années. Nous constatons que la majorité du budget énergétique est allouée au déplacement des données entre la mémoire et les caches. Il est donc primordial d'apporter des solutions matérielles et logicielles permettant de réduire la consommation de ces opérations.

Le cas du supercalculateur Summit. Pour étudier la contrainte énergétique dans l'élaboration d'une plateforme Exascale, nous étudions certaines caractéristiques du supercalculateur Summit présentées dans le tableau 2.3. Ce dernier est classé numéro un au Top500 de novembre 2019 (voir section 2.2.1.1). Il est aussi le premier supercalculateur du classement du Green500 consommant plus de 10 MW (classé 5e en novembre 2019). Pour appréhender l'importance des efforts à fournir pour construire un supercalculateur capable d'exécuter 10^{18} opérations par seconde pour une application réelle, nous comparons l'efficacité énergétique de Summit avec celle d'un supercalculateur Exascale pour l'exécution du benchmark HPL.

Pour fonctionner, Summit a besoin d'une puissance de 10 MW. En novembre 2019, il a atteint une performance de 200 pétaFLOPS (10^{15} FLOPS) lors de l'exécution du benchmark Linpack. Son efficacité énergétique est donc de 20 GFLOP/watt. Un supercalculateur exaflopique consommant entre 20 et 30 MW aurait quant à lui, une efficacité énergétique comprise

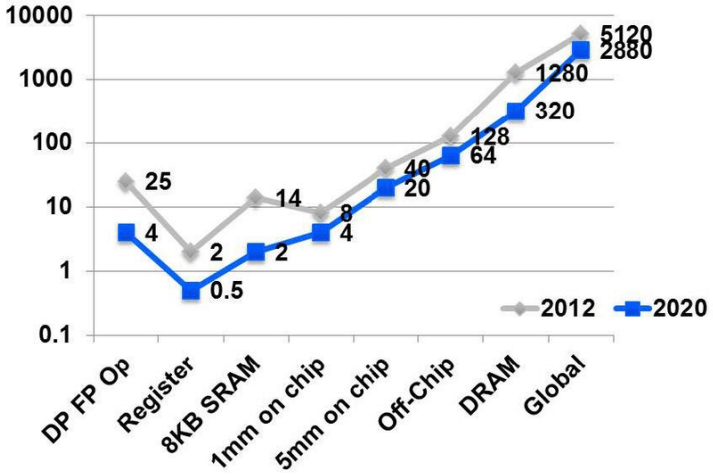


FIGURE 2.28 – Coût énergétique, en picojoules (pJ) par opération à virgule flottante de 64 bits, pour diverses opérations courantes dans un ordinateur. La ligne supérieure (grise) représente le coût énergétique des opérations sur un processeur utilisé en 2012. La ligne inférieure (bleue) prévoit les coûts énergétiques en 2020 [Lel+14].

Supercalculateur	Consommation (MW)	Perf. Linpack (exaFLOPS)	Efficacité énergétique (GFLOP/watt)	Budget énergétique (pJ/opération)	Conso. Linpack (pJ/bit)
Summit	10	0.200	20	50	0.25
Projet Exascale	20 - 30	1	50 - 33	20 - 30	0.1 - 0.15

Tableau 2.3 – Comparaison de l'efficacité énergétique du supercalculateur Summit et de l'efficacité théorique d'une plateforme exascale\cite{Bergman2018}.

entre 33 et 50 GFLOP/watt. Le défi est donc de construire une plateforme 5 fois plus puissante tout en étant deux fois plus efficace énergétiquement. Cependant, le défaut du benchmark Linpack est qu'il ne reflète pas le comportement des applications industrielles. En effet, contrairement aux applications réelles, il ne nécessite aucune communication entre les noeuds, et les instructions sont réalisées sur des données présentes dans le cache. Atteindre un exaFLOPS sur le benchmark HPL sera bien plus facile que d'atteindre la même performance avec une application réelle dont la consommation est principalement due aux déplacements mémoires. Pour mieux appréhender l'impact énergétique des déplacements mémoires, nous calculons l'efficacité énergétique des deux supercalculateurs étudiés en joule/FLOP :

$$\frac{FLOPS}{watt} = \frac{\frac{FLOP}{seconde}}{\frac{joule}{second}} = \frac{FLOP}{joule} \tag{2.7}$$

Nous calculons ainsi le *budget* disponible pour exécuter une opération flottante : 50 pJ pour Summit, contre 20 pJ et 30 pJ pour le supercalculateur Exascale. En moyenne, l'application exécute des instructions vectorielles de 200 bits [22]. L'énergie dépensée par Summit pour calculer un bit est alors de 0.25 pJ. Pour une plateforme exascale, ce budget énergétique serait alors compris en 0.1 et 0.15 pJ/bit.

Un accès à la mémoire se compte en centaine de picojoules par bit et en millier lorsqu'il s'agit de communication entre deux noeuds de calculs. Lorsque l'énergie disponible est de 0.1 pJ/bit et qu'un accès mémoire peut dépasser de cent fois ce budget, il est facile de comprendre que l'essentiel des innovations doivent se porter sur l'amélioration des communications (à l'intérieur des serveurs, mais aussi à l'extérieur) que ce soit par l'utilisation de nouvelles technologies ou d'une restructuration des microarchitectures. Les applications et les algorithmes doivent eux optimiser l'utilisation des données locales pour tirer profit des principes de localité (voir section A.3.2.3).

2.2.4.3 La complexité

Comme présentées dans l'Annexe A, les architectures des ordinateurs ont reçu de nombreuses améliorations au fil des ans. Ces améliorations se sont additionnées et ont permis le développement des processeurs tels que nous les connaissons aujourd'hui : une hiérarchie mémoire, plusieurs coeurs (logique et physique), un pipeline comportant souvent plus de dix étapes, des systèmes d'exécution dans le désordre ou encore de préchargement mémoire.

La complexification est aussi présente dans le logiciel. Pour tirer parti des fonctionnalités offertes par le matériel, plusieurs modèles de programmation sont utilisés pour par exemple exploiter la mémoire partagée et distribuée ou utiliser les différents coeurs d'un processeur. En fonction des tâches à réaliser, plusieurs langages peuvent être utilisés dans une même application (C, C++, fortran, python, cuda...). La programmation **efficace** d'un supercalculateur est ainsi devenue une tâche très difficile. La complexification du matériel et du logiciel a un fort impact

22. Source - https://insidehpc.com/2019/07/flexibly-scalable-high-performance-architectures

sur la performance des applications qui parviennent rarement à exploiter une part significative de la performance disponible. La performance d'une même application peut fortement varier à cause d'une subtilité (la configuration du BIOS, un réglage du système d'exploitation). Certaines technologies (FPGA, GPU) ne sont pas envisagées par des entreprises, car l'adaptation de leurs applications pour fonctionner sur celles-ci est très difficile. Pour s'extraire de la complexité grandissante des plateformes HPC et de leur gestion, de plus en plus d'utilisateurs se tournent vers des solutions dématérialisées telles que le *cloud*.

La sécurité. La complexité du matériel et des couches logicielles représente un grand risque d'exploitation de faille de sécurité. Les applications utilisent de nombreuses librairies qui peuvent présenter des failles, et la difficulté de les mettre à jour (pour la stabilité des applications) peut permettre aux attaquants d'exploiter ces failles. L'accumulation de fonctionnalités a aussi rendu ces architectures sujettes à différentes attaques. En 2018, les chercheurs de Google [Koc+18] ont découvert une faille majeure dans le prédicteur de branchement des processeurs Intel (voir section A.2). Aujourd'hui, les supercalculateurs sont rarement directement connectés au réseau internet, rendant difficile l'attaque de ces plateformes. Cependant, la vision exascale présentée dans le début du chapitre implique la collecte de données et le traitement des données au plus proche de la source de leur création ainsi que l'acheminement de certains résultats vers les centres de calculs, exposant de nombreux lieux d'attaque. Il n'est pas envisageable d'utiliser des voitures autonomes possédant des processeurs, des mémoires, des dizaines de couches logicielles pouvant être attaquées. De même, les données personnelles, telles que les données médicales enregistrées par les montres connectées sont très sensibles.

Passage à l'échelle. Pour obtenir un supercalculateur exascale, il est techniquement possible de construire 5 supercalculateurs de la puissance du Summit et d'agréger leur performance. Cependant, pour les raisons de coût, principalement lié à la consommation énergétique, cette solution n'est pas envisageable. À une telle échelle, le nombre de serveurs serait si important (supérieur à 30000) que des problèmes insoupçonnés à des tailles moindres feraient leur apparition. Les pannes des matériels, les congestions de réseaux et la capacité des applications à utiliser un grand nombre de serveurs impacteraient leur performance rendant le supercalculateur très inefficace. En effet, la programmation parallèle nécessite la synchronisation des ressources à différentes étapes du calcul pour partager des résultats intermédiaires. Ainsi, les pannes matérielles d'une seule ressource de calcul ont de forts impacts sur la performance des applications de calculs parallèles.

2.2.4.4 Les nouvelles technologies

Nous avons montré précédemment que les utilisateurs étaient en demande de puissance de calcul supplémentaire. Le classement du Top500, réalisé tous les 6 mois, permet de classer les plateformes les plus récentes. Les nombreux supercalculateurs construits chaque année rentrent au classement, remplaçant les anciennes plateformes moins performantes. Sur la figure 2.29,

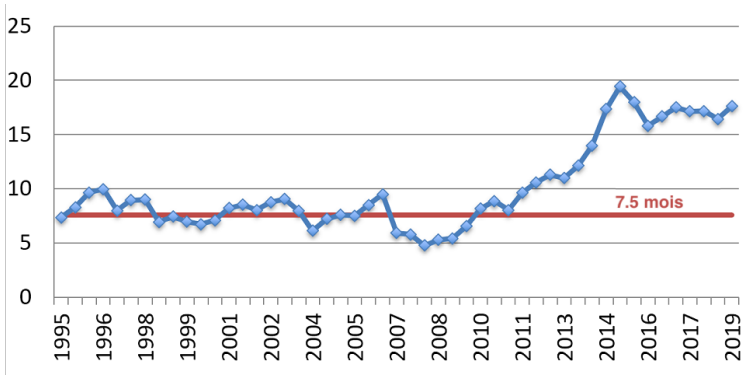
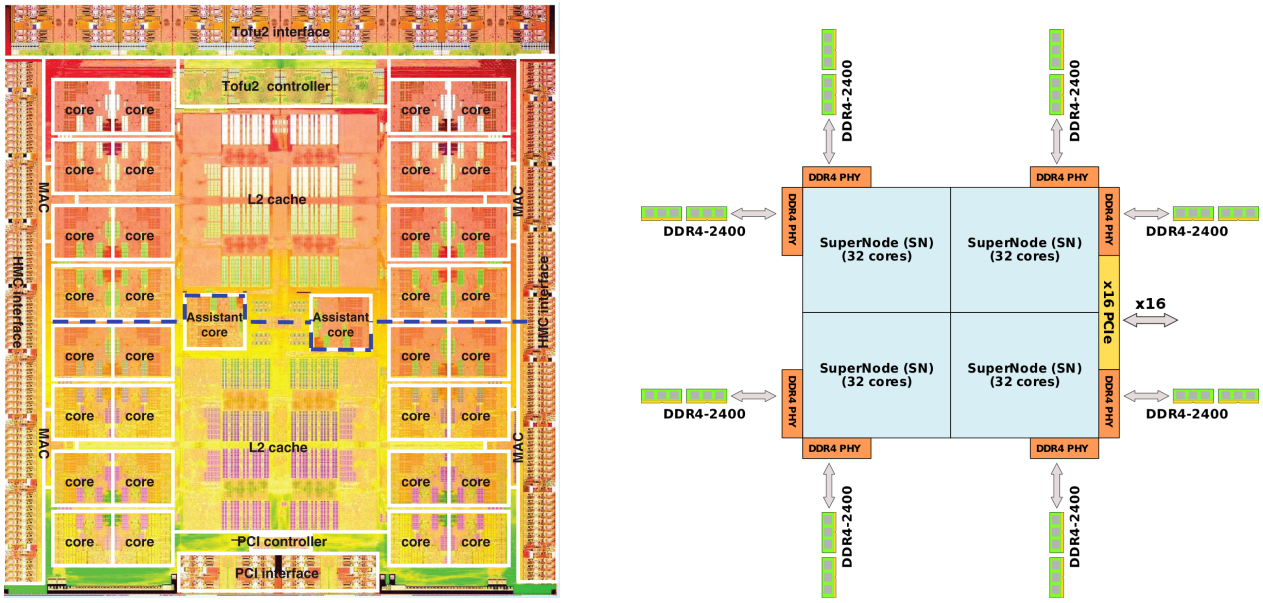


FIGURE 2.29 – Âge moyen des supercalculateurs classés au Top500 [Str18].

nous constatons que l'âge moyen du Top500 a doublé en quelques années indiquant la difficulté à produire des supercalculateurs plus performants. Alors que l'âge moyen du classement oscillait autour de 7 mois pour les 23 premières années du classement, il a dépassé les 15 mois depuis les années 2013. Ces deux constats montrent que les utilisateurs de HPC ne renouvellent plus leurs supercalculateurs, car ils ne trouvent plus de solutions matérielles suffisamment intéressantes pour motiver de nouveaux investissements. De nouvelles technologies doivent émerger pour obtenir de réels gains de performance. L'histoire récente nous a prouvé que ce chemin était le bon avec l'utilisation des GPU (ou d'accélérateurs dédiés comme les TPU de Google) pour accélérer les applications d'apprentissage par machine. Les pressions économiques et énergétiques présentées précédemment nous indiquent que de nombreuses avancées doivent être réalisées dans tous les domaines touchant au HPC : les processeurs, les accélérateurs, les mémoires, les réseaux ou bien la partie logiciel.

Explosion "cambrienne"[23] De nouvelles technologies existent ou sont en cours de développement. Dans la section 2.3.5.4, nous montrons que de nombreuses solutions matérielles, technologiques et logicielles peuvent être utilisées. Des accélérateurs et des mémoires très spécifiques pourront être utilisés. Les utilisateurs de HPC et les architectes doivent alors connaître les besoins de leurs applications pour choisir quelles technologies utiliser (type et quantité de mémoire, utilisation d'accélérateurs). Si ces nouvelles technologies peuvent être une réelle opportunité pour l'élaboration de plateformes plus puissantes et plus efficaces, elles sont aussi un défi. En effet, jusqu'à récemment, la majorité des plateformes étaient construites de façon similaire : un processeur x86 pouvant être associé à un GPU (voir section 2.3.4.3). Si les applications obtenaient de faibles performances, il était difficile de s'orienter vers d'autres solutions alors inexistantes. La majorité des supercalculateurs sont utilisés par différents utilisateurs et exécutent différents types d'applications. Le choix des technologies utilisées doit alors prendre en compte ces différentes applications pour construire la plateforme la plus efficace possible.

23. L'explosion cambrienne désigne l'apparition soudaine (à l'échelle des temps géologiques) de la plupart des grands embranchements actuels des animaux pluricellulaires ainsi qu'une grande diversification des espèces animales, végétales et bactériennes (Wikipédia).



(a) Le processeur Sparc64 est produit par Fujitsu et utilise une mémoire empilée HMC[Yos+15].

(b) Le coprocesseur Matrix 2000 possède 128 coeurs et 8 canaux mémoires.

FIGURE 2.30 – De nouvelles architectures sont utilisées dans les supercalculateurs classés au Top500.

Souvent ces centres de calculs seront hétérogènes avec différents processeurs.

Nouveautés au Top500. En étudiant le classement du Top500, nous constatons les premiers signes de l'évolution des plateformes qui utilisent de nouveaux types de processeurs ou d'accélérateurs (voir figure 2.30). En novembre 2019, trois supercalculateurs classés à la 84e, 72e et 116e place étaient équipés de processeur Sparc64 XIfx [Yos18]. Ce processeur est produit par Fujitsu et possède 34 coeurs (dont deux réservés en priorité au système d'exploitation et à la gestion des communications). Il possède 24 Mb de cache L2 et atteint une performance de 1,1 téraFLOPS (voir figure 2.30a). Pour atteindre de hautes performances sur des applications réelles, le processeur est doté d'une mémoire empilée HMC de 32 GB [GY17]. Concernant les accélérateurs, nous pouvons citer le coprocesseur Matrix-2000 qui a permis de doubler la puissance du supercalculateur chinois Tianhe-2 en remplaçant les anciens accélérateurs KNL par une nouvelle architecture développée exclusivement pour lui. Cet accélérateur 64 bits produit par NUDT, possède 128 coeurs RISC à 1,2 GHz permettant d'atteindre une performance crête de 4,92 téraFLOPS pour une enveloppe thermique de 240W. Quatre groupes de 32 coeurs sont reliés à la mémoire avec un total de 8 canaux mémoire (voir figure 2.30b).

2.2.4.5 Rapidité de l'évaluation des nouvelles technologies

Face à l'arrivée de ces nombreuses technologies, les entreprises doivent rapidement être capables d'évaluer leurs qualités et leurs défauts et quantifier leur adéquation avec leurs appli-

cations. Les évolutions récentes sont très rapides avec le développement de dizaines de modèles de mémoire, de nouveaux accélérateurs et de processeurs. Certaines technologies encore en développement sont déjà dépassées par l'annonce de nouvelles technologies.

Une technologie émergente, très performante pour une application, avec des consommations énergétiques ou un coût très faible peut permettre à une industrie de faire un bond de performance dans un domaine. Grâce à ces technologies, l'économie d'un domaine peut être fortement modifiée en permettant de réaliser des simulations dix fois plus rapidement ou en divisant leur coût du même facteur.

La vitesse de décision et d'utilisation d'une nouvelle technologie dans notre société mondialisée est cruciale. Les entreprises doivent donc disposer de moyens (techniques et financiers) pour se tenir à l'état de l'art des technologies. Le développement de partenariat (avec des universités par exemple) est une des clefs permettant de s'assurer de la capacité des entreprises à ingérer ces nouvelles technologies.

2.2.4.6 L'expertise

Pour construire les plateformes de demain, il est nécessaire de pouvoir qualifier les nouvelles architectures émergentes. Cette qualification devant se faire rapidement sur des dizaines d'architectures différentes, il est important pour les acteurs du HPC d'avoir les moyens adéquats pour mener à bien cette mission. Pour motiver les choix à entreprendre, une modélisation économique des performances ($FLOP/\$$, $GB/s/\$$) doit pouvoir être réalisée. Une fois les architectures sélectionnées, les programmeurs doivent être capables d'extraire une part importante des performances disponibles. Les architectures ciblées pouvant être très différentes des processeurs x86 largement utilisés aujourd'hui, les programmeurs doivent avoir de solides connaissances et les outils adaptés, sans quoi certaines architectures devront être abandonnées. Les outils à leur disposition doivent leur permettre d'établir un profil des besoins de son application pour pouvoir caractériser et sélectionner les différentes plateformes. Pour étudier son application et valider leur performance une fois le code porté, il est nécessaire d'avoir des outils facilement utilisables sur différentes plateformes (et différentes microarchitectures).

Les FPGA (Field Programmable Gate Array) en sont un bon exemple. Cette technologie permet de faire des périphériques très performants et très efficaces en termes de consommation électrique. L'idée principale étant de laisser au programmeur le développement complet du circuit électronique pour qu'il corresponde parfaitement à son besoin. Mais la programmation de tels circuits est très complexe et demande des mois, souvent des années pour des codes industriels, pour être réalisée. Ainsi, malgré l'efficacité, prouvée, de cette technologie, les entreprises ne s'y lancent pas à cause des coûts engendrés par l'achat du matériel et la taille des équipes requises pour programmer et supporter les applications.

Optimisation des codes. Le défi de l'exascale ne vient pas seulement du matériel et des technologies que nous allons utiliser comme le soulignent certains travaux [Bar+12]. Il nécessite aussi de repenser une grande partie des logiciels. Cette partie est très délicate, car les

Year	*Method*	*Reference*	*Storage*	*Flops*
1947	GE (banded)	Von Neumann & Goldstine	n^5	n^7
1950	Optimal SOR	Young	n^3	$n^4 \log n$
1971	CG	Reid	n^3	$n^{3.5} \log n$
1984	Full MG	Brandt	n^3	n^3

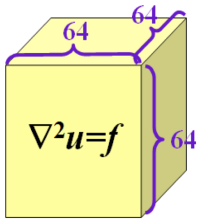


FIGURE 2.31 – L'utilisation de nouvelles méthodes mathématiques a permis de réduire de plusieurs facteurs la quantité de stockage (*storage*) et le nombre de calculs nécessaires (*flops*) pour la résolution d'une équation de poisson [24].

applications atteignent souvent plusieurs dizaines de milliers de lignes de codes. Changer un compilateur ou un debugger peut alors s'avérer plus complexe qu'il n'y paraît. Pour atteindre les meilleures performances possibles, les programmeurs doivent être capables d'écrire des applications optimisées pour les architectures utilisées. Pour cela, il est nécessaire de restructurer le code ou d'utiliser d'autres algorithmes pour tirer parti des caractéristiques du matériel (hiérarchie de cache, taille de la mémoire, nombre de coeurs...). Des optimisations telles que le découpage en bloc [Xue12] ou le *time squewing* [Won02] existent et ont prouvé leur efficacité. Cependant, leur implémentation sur des applications industrielles peut être très fastidieuse et les performances atteintes peuvent être différentes de celles attendues.

Adapter les algorithmes. Une source (presque) inépuisable d'accélération des applications vient de l'utilisation de nouveaux algorithmes. La figure 2.31, montre comment l'utilisation de nouvelles méthodes mathématiques permet d'accélérer de plusieurs facteurs la résolution d'une équation de poisson. Une application nécessitant 6 mois de calculs en 1947 peut être résolue grâce à une méthode multigrille [Bra82] en moins d'une seconde.

Pour réduire le nombre d'opérations lors de la multiplication de matrices pour les algorithmes d'apprentissage par machine, plusieurs techniques peuvent être utilisées [Sze+17] : la transformée de Fourier rapide [Vas+14], l'algorithme de Strassen [CX14] ou encore l'algorithme de Coppersmith-Winograd [Li+16]. En fonction des caractéristiques de l'architecture (stockage disponible), du besoin de stabilité numérique, ou de la taille des matrices utilisées, chaque méthode a ses avantages et inconvénients. Le compilateur peut alors choisir la meilleure technique à utiliser en fonction de ces paramètres.

Pour les plateformes exascale, les algorithmes doivent être adaptés à certaines contraintes. Le coût des déplacements des données, en énergie et en temps, doit être le moteur de leur développement et de leurs améliorations. En effet, le coût énergétique de l'exécution d'une opération flottante peut être considéré comme gratuit quand on le compare au coût du déplacement des

24. Tableau extrait de la présentation de David Keyes - `http://www.cs.odu.edu/~keyes/talks/SC_IMI.ppt`

données. La plus grande pénalité venant des communications entre les noeuds de calculs, les algorithmes doivent réduire au maximum ces communications en découpant stratégiquement le jeu de données. Il peut même être avantageux de recalculer certains résultats localement, plutôt que de les communiquer entre deux serveurs. L'algorithme de Strassen [Lip+12] permet par exemple la multiplication de deux matrices sans échanges de données. Ces techniques ont un gain double : elles sont plus rapides, car l'exécution n'est pas pénalisée par la performance du réseau, et elles sont plus efficaces énergétiquement, car elles ne payent pas le coût de la communication des données. Cependant, ces méthodes peuvent entraîner des problèmes de stabilité numérique [Kha13].

Challenge. La complexification des applications et du matériel ajoutée au manque de connaissances et d'outils des utilisateurs se traduit par une mauvaise utilisation des plateformes évoquée dans la section 2.2.1.1. Bien que la recherche ait réalisé de nombreuses découvertes, peu d'entre elles sont réellement appliquées sur les codes industriels. La loi de Moore assurant une évolution constante des performances des architectures, l'analyse et l'optimisation des codes ont été laissées de côté. Les investissements réalisés dans le développement des matériels se sont faits au détriment des logiciels. Aujourd'hui, nous constatons un manque de connaissances fines des architectures dû à leur complexité croissante ainsi que le manque d'outils adéquats pour réaliser les optimisations nécessaires.

2.3 Opportunités pour le développement de nouveaux supercalculateurs

Nous avons montré dans les sections précédentes que poursuivre la stratégie qui consistait à ajouter toujours plus de serveurs pour construire un supercalculateur n'est plus viable. Nous avons présenté les principaux défis que les utilisateurs de HPC doivent relever et les barrières rencontrées. Pour y parvenir, des évolutions technologiques majeures doivent être faites et l'architecture des plateformes doit être repensée. Ce constat est partagé par les auteurs du rapport de Pathforward [Luc+14] : "*Les améliorations considérables nécessaires à la réalisation d'un supercalculateur exascale ne seront pas satisfaites par des améliorations progressives des techniques conventionnelles actuelles.*" Si les défis sont nombreux et difficiles à relever, de nombreuses opportunités sont disponibles. Dans cette section, nous présentons les opportunités principales à considérer : l'apparition de nouvelles technologies très différentes de celles actuellement utilisées et la reconsidération complète de l'architecture des plateformes.

2.3.1 Investissements financiers

La première condition pour réaliser les nombreux développements technologiques nécessaires est la présence d'investissements financiers. L'industrie du HPC investit massivement dans le développement de nouvelles technologies pour répondre au besoin de puissances de calcul. Une

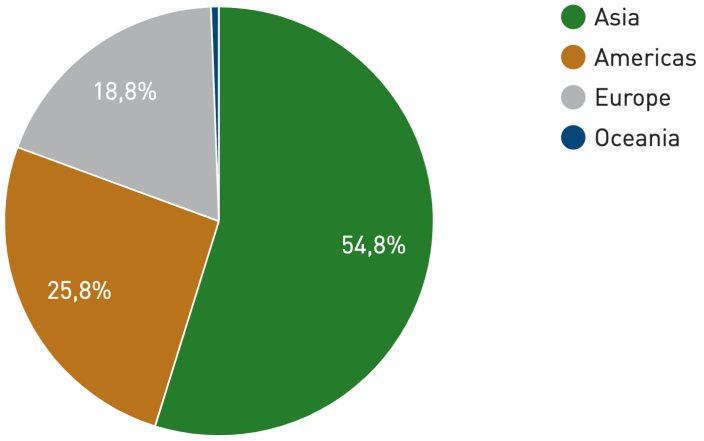


FIGURE 2.32 – Répartition de la puissance de calcul du Top500 de novembre 2019 entre les continents.

analyse de marché paru en 2016 montre que le marché du High Performance Computing pèsera 36 milliards de dollars en 2020, alors qu'il en valait 28 en 2015 [25].

Si le développement des plateformes exascales est motivé par son aspect financier pour les industries, la construction du premier supercalculateur exascale est un enjeu politique pour les nations. Comme le fut autrefois la course à l'espace et la conquête de la lune, les nations ont à coeur d'être les premières à obtenir une telle architecture et s'en donnent les moyens. L'obtention d'une telle performance est un symbole qui mettra en lumière les acteurs qui y parviendront en premier (nations, universités, constructeurs).

Investissements Européens Dans le classement du TOP500 paru en novembre 2017, aucun des 10 supercalculateurs les plus puissants et seulement 5 des 20 meilleurs supercalculateurs mondiaux sont installés en Europe. Alors que l'Asie et l'Amérique se partagent près de 80% de la puissance de calcul mondiale, l'Europe n'en possède que 18.8% sur son territoire (voir figure 2.32).

Pour accélérer le développement de ses infrastructures, l'Europe a financé un grand projet nommé Horizon 2020. Horizon 2020 est le plus grand programme de recherche et d'innovation jamais mis en place par l'Union européenne. Ce projet d'envergure prévoit d'investir 79 milliards d'euros de 2014 à 2020 pour développer trois piliers : l'excellence scientifique, la primauté industrielle et les défis sociétaux. En janvier 2018, la Commission européenne a annoncé investir 1 milliard dans un partenariat réalisé entre le domaine public et privé nommé EuroHPC [Eur18]. Les objectifs d'EuroHPC sont de développer et déployer une infrastructure de calcul de classe mondiale. Cette entreprise communautaire aura pour objectif de construire 6 supercalculateurs : 2 supercalculateurs petascale (10^{16} opérations) et deux infrastructures "*pré exascale*" en 2021. Les architectures pré exascale vont permettre d'utiliser les premières ver-

25. http://www.marketsandmarkets.com/Market-Reports/Quantum

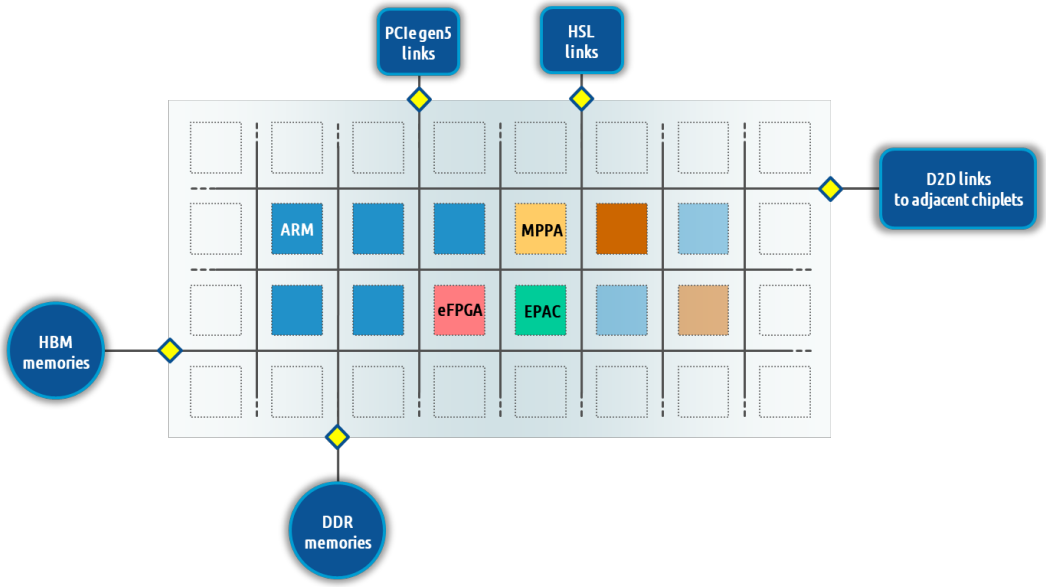


FIGURE 2.33 – Les processeurs GPP utiliseront un réseau à mailles 2D sur la puce permettant de relier différents types de coeurs : ARM-V, FPGA embarqué (eFPGA) pouvant être reprogrammé, un processeur vectoriel (MPPA).

sions des processeurs développées par EuroHPC. Pour assurer son indépendance technologique, le projet EPI (*European Processor Initiative*) a pour objectif de développer de nouveaux processeurs et des accélérateurs MPPA (*massively parallel processor array*) produits par Kalray à Grenoble (voir figure 2.33). Enfin, le supercalculateur exaflopique européen prévu entre 2022 et 2023 pourra compter sur de nouvelles architectures, très efficaces énergétiquement, basées sur des processeurs ARM actuellement en développement. Une des motivations étant de réduire sa dépendance aux autres pays pour être plus compétitif.

Les États-Unis. En 2017, le département de l'énergie américain a annoncé le financement d'un projet de 232 millions d'euros nommé PathForward. Ce budget est réparti entre 6 entreprises américaines (HPE, AMD, Cray, IBM, Intel et Nvidia) qui doivent ajouter 40% de financement supplémentaire pour un total de plus de 400 millions d'euros. L'objectif de ce projet est de construire le premier ordinateur exascale en 2021 en finançant les développements matériels et logiciels nécessaires.

2.3.2 Nouvelles technologies mémoire

Dans cette section, nous utilisons le terme *gap* pour traduire un écart de performance entre deux technologies. Le terme "memory gap" est couramment utilisé dans la littérature pour désigner l'écart de performance actuel entre les processeurs et le système mémoire [Wil01].

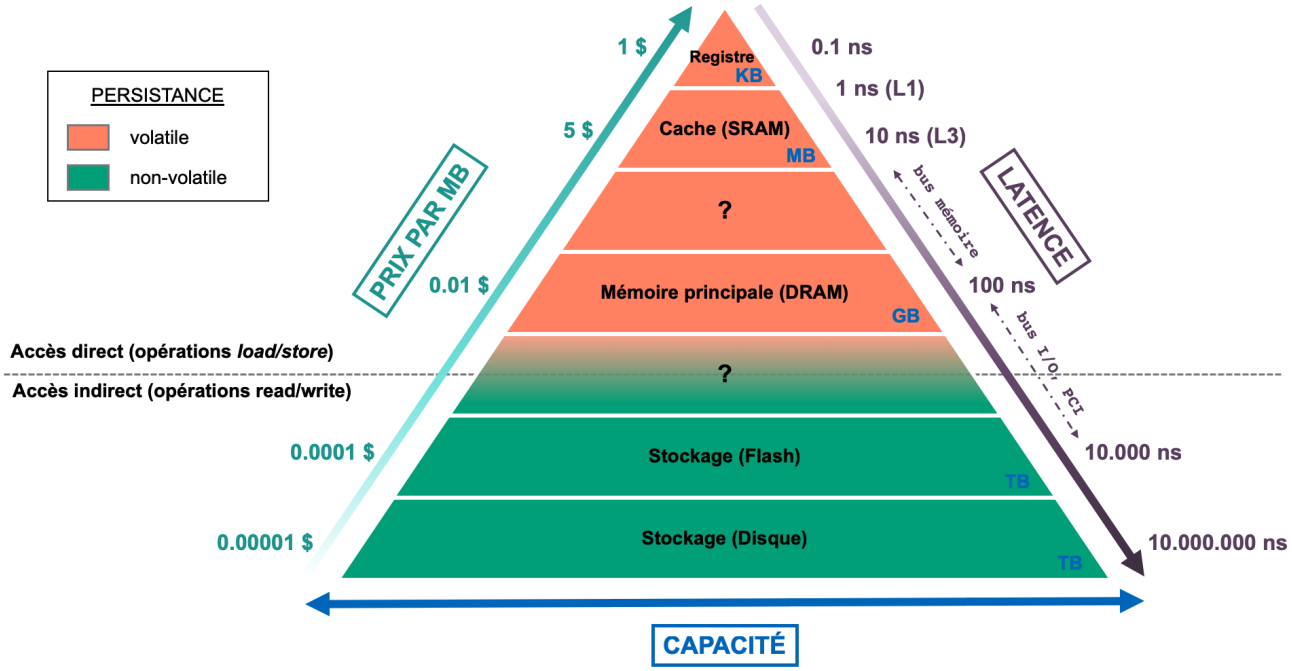


FIGURE 2.34 – Hiérarchisation des différents types de mémoire en fonction du coût, de la latence et de la capacité de stockage habituellement utilisés dans les architectures modernes. Les écarts de performance relevés dans la hiérarchie mémoire sont modélisés par un '?'.

2.3.2.1 Memory gap : état des lieux des technologies

L'évolution des capacités opérationnelles des processeurs et de la performance des mémoires (débits, latence, taille) a été très inégale. La figure 2.34 présente sous forme de pyramide les différentes technologies mémoire actuellement utilisées dans le système. Les trois principaux facteurs différenciant les différentes technologies mémoire sont : le prix, la latence d'accès et la capacité (densité). Pour mieux appréhender la différence de latence des technologies, nous les convertissons dans un temps plus facilement appréciable pour un humain. Si un accès au cache L1 prenait 3 secondes, un accès mémoire prendrait alors plus de 5 minutes. Si la donnée à récupérer se trouve sur un disque SSD, il faudrait attendre 3 heures avant de la recevoir. Si celle-ci se trouve sur un disque optique, il faudrait alors patienter 10 mois. Il est facile de comprendre pourquoi le *manque* (*miss*) d'une donnée dans le cache pénalise énormément la performance d'une application.

Malgré le développement d'une hiérarchie mémoire (voir section A.3) pour réduire ces écarts de performance, nous remarquons la présence de deux *trous* (*memory gap*) de performance. Pour résoudre ce problème, l'industrie poursuit le travail débuté avec l'introduction de la hiérarchie mémoire grâce à deux solutions :

— Réaliser le traitement directement dans la mémoire grâce aux techniques de calcul en mémoire (*processing-in memory* (PIM)) [Sin+19]. En déplaçant le calcul en mémoire, les latences d'accès sont réduites et les débits augmentés. Des implémentations d'archi-

tecture PIM utilisent des mémoires SRAM qui ont permis d'accélérer des algorithmes d'apprentissage machine [ZWV16 ; BC18 ; Kan+18]. D'autres implémentations utilisent de la mémoire DRAM [Ses+17] [Li+17] permettant de réaliser des opérations logiques `AND` et `OR` sur des barrettes mémoires DRAM classiques non modifiées [GTW19]. Des technologies comme le memristor permettent de réaliser des multiplications de matrices dont chaque valeur peut être calculée simultanément. Ce composant électronique a été décrit par Leon Chua en 1971 dans l'article " *Memristor - The Missing Circuit Element*" [Chu71]. La première implémentation physique a été réalisée par une équipe de recherche des laboratoires d'HP conduite par Stanley Williams en 2008 et publiée dans l'article "*The missing memristor found*" [Str+08]. Cette mémoire permet d'encoder un nombre réel sur un seul bit. L'information encodée peut ainsi avoir une infinité de valeurs contrairement à deux valeurs pour les bits des mémoires utilisées habituellement.

— La deuxième solution permettant d'accélérer les accès mémoire est de combler les deux *trous* de performance constatés sur la figure 2.34 grâce au développement de nouvelles technologies mémoire. Les différentes solutions envisagées sont discutées dans le reste de cette section.

2.3.2.2 Trou entre SRAM et DRAM

Pour réduire le trou séparant les mémoires cache et la mémoire centrale, les processeurs ont vu leur dernier niveau de cache s'agrandir. Cependant, la mémoire SRAM utilisée est très chère à produire et sa densité ne permet plus d'agrandir la capacité des mémoires cache. Côté mémoire principale, le développement de différentes technologies mémoire (DDR3, DDR4, DDR5) ne permet pas de faire évoluer fortement la performance des applications. La raison principale vient de la difficulté à augmenter le débit du bus mémoire qui est limité par le nombre de broches utilisables sur le processeur. La figure 2.35 montre l'évolution de la proportion de broches d'un processeur qui est affectée au bus mémoire. Il est aujourd'hui très difficile d'allouer plus de broches pour le bus mémoire. Le nombre de canaux a augmenté, permettant d'améliorer le débit du bus mémoire. Cependant, le nombre de coeurs a lui aussi évolué, bien plus rapidement que le nombre de canaux mémoire. Ainsi, la bande passante mémoire par coeur est passée de 6.4 Gb/s pour un processeur de 8 coeurs en 2012 à 4.6 Gb/s pour un processeur de 28 coeurs en 2016.

À partir de ce constat, et celui fait dans la section 2.2.4.2 concernant la consommation énergétique du système mémoire, les architectures ont été repensées pour placer la mémoire directement sur les puces (*On Package Memory*) au plus proche des circuits de traitement (*near-memory processing*). En plaçant la mémoire directement sur la puce du processeur, les limitations imposées par le bus mémoire (débit, consommation électrique) peuvent être contournées notamment grâce à l'utilisation de bus plus large (voir figure 2.36a). Afin de pouvoir installer des espaces mémoire suffisamment larges directement sur la puce, un nouveau type de mémoire a été développé : les mémoires 3D. La principale différence avec la mémoire conventionnelle est son architecture qui est en 3D. Alors que la mémoire DRAM s'étale sur deux dimensions X, Y, la mémoire 3D s'étend en plus sur une troisième dimension Z en s'empilant

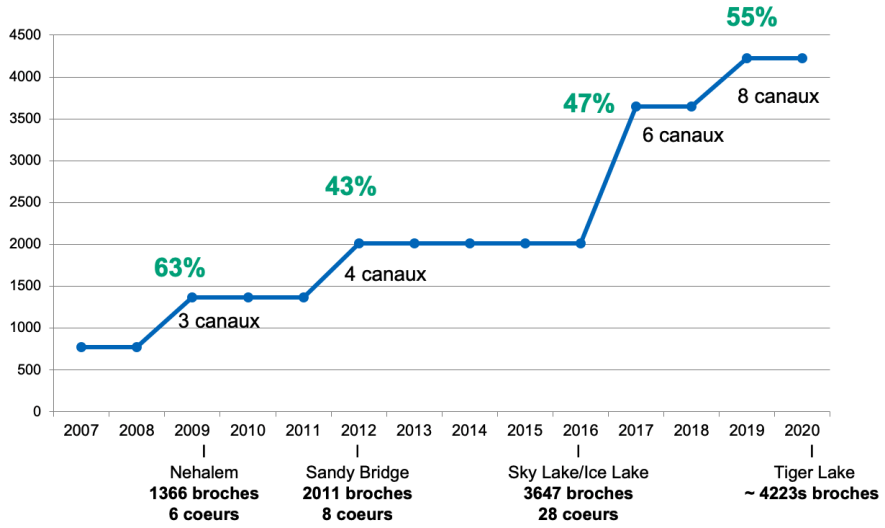
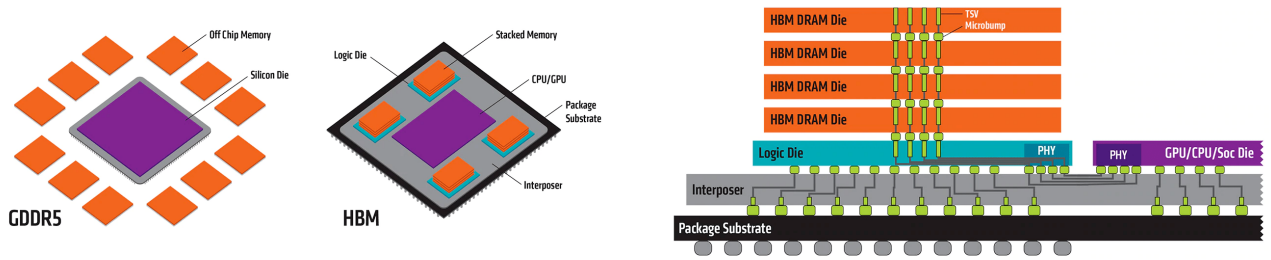


FIGURE 2.35 – Nombre et proportion de broches allouées aux bus mémoires des dernières architectures de processeur Intel.



(a) Pour atteindre de meilleures performances, les mémoires 3D sont placées directement sur la puce.

(b) Les mémoires 3D sont un empilement de mémoire DDR relié à un interposeur grâce à des *vias* (TSV, TCI).

FIGURE 2.36 – Architecture et utilisation d'une mémoire 3D sur un GPU [26]

(*stack*, voir figure 2.36b). En empilant des couches de silicium, ces mémoires sont très denses. Ainsi, 1GB de GGDR5 qui nécessite $672mm^2$, n'a besoin que de $35mm^2$ pour la même capacité de mémoire 3D, soit une réduction de 94%.

Actuellement, les mémoires 3D sont principalement commercialisées sous les technologies HBM (*High Memory Bandwidth* [Sta13]) produite par AMD, Samsung et Hynix et de la HMC (*Hybrid Memory Cube*[JK12]) produite par Micron. Les mémoires HBM se composent actuellement de quatre puces DRAM sur une puce de base et possèdent deux canaux de 128 bits par puce DRAM, soit 8 canaux au total, ce qui donne 1024 bits par pile d'interfaces mémoire (voir

25. Mesure réalisée par AMD sur 1GB GDDR5 (4x256MB ICs) et 1GB HBM-2 (1x4-Hi)
26. Source AMD - https://www.amd.com/fr/technologies/hbm

	Largeur bus	Capacité (GB)	Débit (Gb/s)	Fréquence mémoire	Bande passante (Gb/s)	Consommation (pJ/bit)
GDDR5	768	24	8	1.25 GHz	480	14
HBM2	4096	8	2	1 GHz	1024	4

Tableau 2.4 – Comparaison des mémoires GDDR5 utilisées sur les GPU NVidia K80 et des mémoires HBM2 utilisées sur un GPU utilisant 4 piles de mémoire.

tableau 2.4). Une carte graphique possédant 4 piles possède ainsi un bus mémoire de 4096 bits. Pour améliorer l'efficacité énergétique, la fréquence des mémoires 3D est moins rapide que celle des mémoires DRAM conventionnelles. De plus la proximité de ces mémoires dites *on-package*, permet de réduire la distance de communication avec les processeurs et de réduire la consommation électrique (3.9 pJ/bit pour la HBM2 contre 14 pJ/bit pour la GDDR5). Malgré une fréquence plus faible, le large bus mémoire directement connecté à la puce permet d'atteindre des bandes passantes plus élevées. Pour une enveloppe énergétique de 60W, une mémoire GDDR5 est capable de fournir un débit de 536 Gb/s quand une mémoire HBM2 peut atteindre 1.9 Tb/s [OCo+17]. Les mesures de performances réalisées à l'aide du benchmark Stream ont permis de mesurer des écarts de performances d'un facteur 4 avec une mémoire DDR4 classique[Pen+17], tout en consommant 50% d'énergie en moins . Cependant, des latences d'accès réduites (15%) ont été mesurées et peuvent impacter la performance de certaines applications. L'utilisation de nombreux coeurs et d'instructions de préchargement mémoire peuvent cependant permettre d'atteindre de meilleurs résultats [Pen+17].

2.3.2.3 Trou entre DRAM et Flash

La nécessité d'augmenter les capacités de stockage, la pression énergétique et la faiblesse d'évolution des performances du système mémoire sont à l'origine du développement de nouvelles technologies visant à combler le trou situé entre la mémoire et le stockage. L'utilisation de ces nouvelles mémoires n'est pas seulement une évolution de performance. Elles vont aussi permettre de développer de nouvelles approches logicielles. Par exemple, lors de l'arrêt d'un ordinateur, si la totalité de la mémoire y est sauvée, le temps de redémarrage sera accéléré. Les applications telles que les bases de données relationnelles sont développées pour anticiper les longues latences des disques pourront être stockées directement en mémoire, sans risque de perdre l'information (*In memory database* [Ouk+15]). Les applications de simulations numériques pourront rapidement générer des points de contrôle (*checkpoint*), permettant de reprendre le traitement suite à une erreur.

L'objectif est donc de développer de nouvelles mémoires ayant les avantages des deux technologies (DRAM et flash) sans leurs inconvénients. Plusieurs critères sont alors essentiels pour leur adoption ([FW08]). Le premier concerne la persistance des données, permettant entre autres de réduire leur consommation énergétique. Pour des raisons de fiabilité et de densité, ces mémoires ne doivent pas contenir de parties mobiles telles qu'un disque rotatif. Pour être utilisées comme mémoire, ces nouvelles technologies doivent pouvoir atteindre des latences d'accès faibles (au-

26. Information NVIDIA - https://www.extremetech.com/extreme/226240-sk-hynix-highlights

tour de 200 ns[IBM13]), proches de celle de la DRAM. Les technologies développées doivent être endurantes pour pouvoir être utilisées intensivement (entre 10^9 et 10^{12} écritures par cellule [IBM13]). À terme, ces technologies devraient remplacer les mémoires DRAM, elles doivent donc être adressables par octet. Enfin, pour être accepté par l'industrie, le prix de ces mémoires doit être compétitif avec les technologies existantes. Avec les critères exposés précédemment, nous constatons qu'il n'est pas possible de compter sur les technologies existantes :

— Les disques optiques permettent d'obtenir de grandes capacités de stockage. Cependant, à cause des parties mécaniques (disques rotatifs, bras de lecture) les latences et les débits sont trop faibles. De plus, pour obtenir des latences acceptables, les disques doivent tourner continuellement, augmentant leur consommation énergétique et réduisant leur fiabilité.

— La technologie flash ne possède pas une endurance aux écritures suffisante (10^6 écritures), loin des objectifs fixés (10^9 écritures). De plus, par héritage de la technologie des disques optiques, les disques SSD ne sont adressables que par bloc.

— La mémoire DRAM possède de très bonnes performances en lecture comme en écriture. Cependant, sa densité est faible et elle doit constamment être alimentée pour conserver ses données, augmentant la consommation électrique.

NVM, NVRAM ou SCM ? De nombreux termes sont utilisés dans la littérature pour désigner ces nouvelles mémoires : NVM (*non-volatile memory*), NVRAM (*non volatile RAM*) ou encore PM (*persistent memory*). Le terme NVM est souvent confondu avec le terme NVMe qui désigne un nouveau protocole de communication visant à remplacer le protocole SATA. Les différentes implémentations des NVRAM sont présentées sur la figure 2.37. Le premier critère de classement est celui de la volatilité puis de l'adressabilité. La seule NVRAM couramment utilisée qui n'est adressable qu'en bloc est la mémoire flash, ne répondant ainsi pas au critère d'adressabilité fixé ci-dessus. Il y a ensuite deux façons d'implémenter des NVRAM adressables par octet : les NVDIMM (Non-Volatile DIMM), les SCM (*Storage Class Memory*).

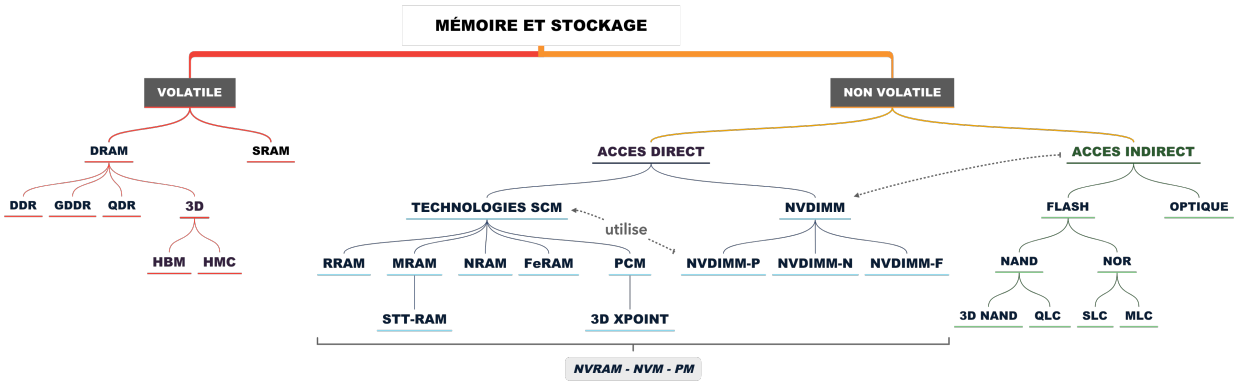
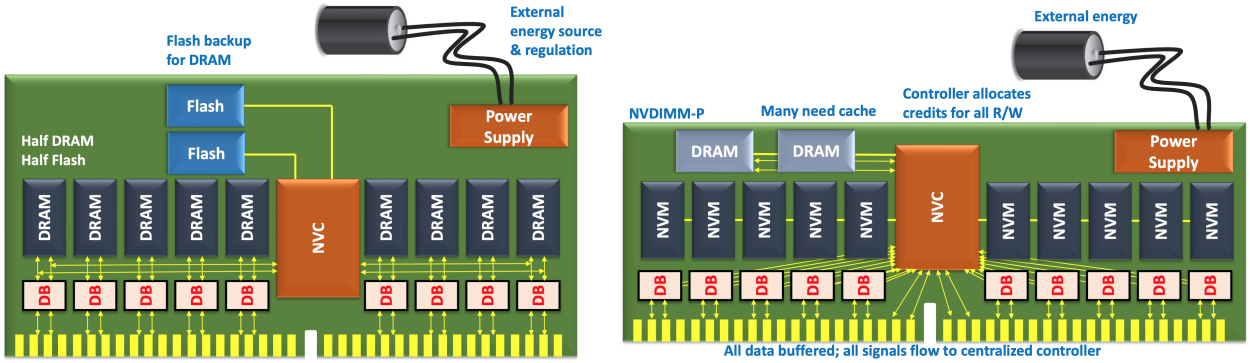


FIGURE 2.37 – Tri des technologies de stockage en fonction de la volatilité et l'adressabilité.



(a) NVDIMM-N (b) NVDIMM-P

FIGURE 2.38 – Les deux standards NVDIMM développés par l'organisation JEDEC [28].

Les NVDIMM. La NVDIMM est une technologie mémoire qui utilise le même format que les barrettes de mémoires classiques DRAM [Chr17], elle peut donc être adaptée sur les serveurs actuels. Le système d'exploitation doit cependant être adapté pour profiter des avantages de la persistance et être, par exemple, capable de redémarrer directement à partir des données se trouvant en mémoire (les versions supérieures à Linux 4.4 sont compatibles). L'organisation JEDEC [27] a été créée en 1958 et développe, entre autres, les standards utilisés en micro électronique. L'organisation a publié 3 standards pour le développement des NVDIMM (voir figure 2.38).

Le premier standard (Type 1) appelé NVDIMM-N (voir figure 2.38a) utilise de la mémoire DRAM associée à de la mémoire flash pour permettre la persistance des données ainsi qu'une source d'énergie indépendante permettant la sauvegarde des données en cas de coupure électrique. Lorsque le serveur est de nouveau alimenté, les données sont transférées de la mémoire flash à la DRAM pour permettre un redémarrage rapide. Pour l'utilisateur, la mémoire flash est invisible et ne peut pas être adressée. La capacité de ces mémoires aura tendance à être faible, car elles dépendent de la densité de la mémoire flash. Cependant, si la mémoire DRAM est bien utilisée, elle permet d'obtenir des performances et une endurance aux écritures proches d'une barrette de mémoire classique et bénéficie de la persistance grâce à la mémoire flash.

Le second standard (Type 3) appelé NVDIMM-F utilise seulement de la mémoire flash et est présenté au système d'exploitation comme un stockage. Cependant, contrairement à un disque classique, il bénéficie des performances du bus mémoire. Ce standard a depuis été abandonné.

Le troisième standard (Type 4) appelé NVDIMM-P (voir figure 2.38b) utilise de la mémoire DRAM associée à des technologies mémoire SCM (voir paragraphe suivant). Ces barrettes utilisent des banques de mémoires DRAM comme cache pour accélérer les communications avec les modules SCM. Le standard prévoit la possibilité d'adresser la mémoire en octets ou en blocs permettant de les utiliser comme mémoire, ou comme stockage. Les performances et la capacité

27. JEDEC - https://www.jedec.org/
28. Source des illustrations : Bill Gervasi (Nantero) - https://www.hotchips.org/hc30/2conf/2.04_Nantero_20180818_hotchips_gervasi_nram_presentation.pdf

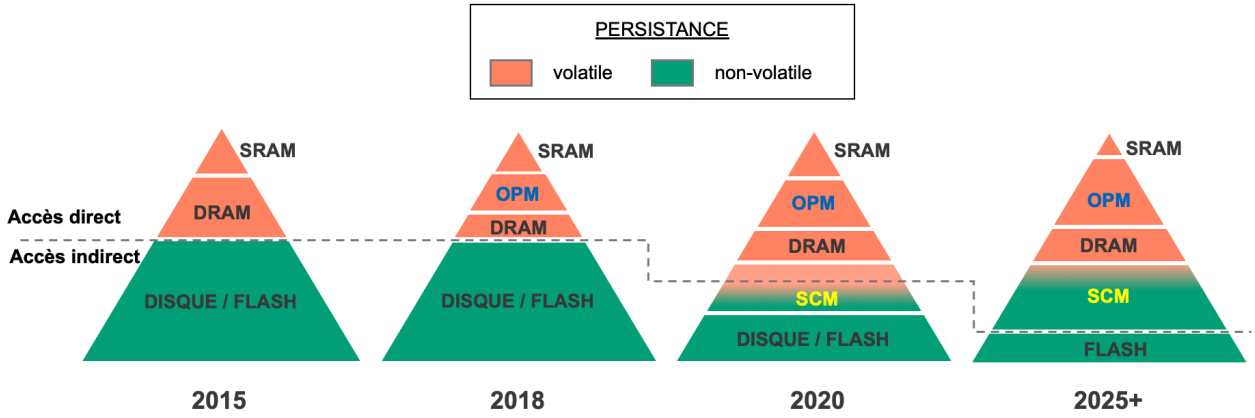


FIGURE 2.39 – Évolution du système mémoire : les mémoires OPM (On Package Memory) telles que la HBM et les mémoires SCM (Storage Class Memory) vont permettre de compléter les trous de performance respectifs entre la SRAM et la DRAM ainsi qu'entre la DRAM et la mémoire flash.

de ce format de barrettes dépendent essentiellement des technologies SCM utilisées. C'est un avantage de ce type de NVDIMM, la technologie et le mode d'adressage utilisés pourront être adaptés en fonction des besoins des applications. Bien que la mémoire SCM soit non volatile, la barrette mémoire nécessite une alimentation pour sauvegarder le contenu de la DRAM sur la SCM en cas de coupure électrique.

Les Storage Class Memory (SCM) Les mémoires SCM regroupent toutes les nouvelles technologies développées répondant aux critères précédemment cités. La mémoire flash peut être considérée comme une SCM. Cependant, dans la littérature, le terme SCM est généralement utilisé pour désigner des technologies très innovantes aux caractéristiques supérieures à la mémoire flash. Cette technologie possède en effet quelques inconvénients qui ne permettront pas de l'utiliser comme une mémoire. Tout d'abord, ses performances asymétriques dont l'écriture est dix fois plus lente que la lecture. L'écriture et l'effacement ne peuvent se faire que par bloc. Enfin, l'endurance de la mémoire flash ne permet pas de supporter suffisamment d'écriture par cellule. En fonction des technologies utilisées (SLC, MLC, TLC), le nombre d'écritures par cellule est compris entre 10^3 et 10^5.

Dans la suite de cette section, nous présentons les technologies SCM ayant le plus de potentiel pour être industrialisées. En fonction des utilisations, certains types de mémoires seront préférés. Pour des applications de stockage, leur coût devra être proche de celui des disques, mais pourra avoir des latences plus élevées (inférieur à la microseconde) et des débits plus faibles (en centaines de mégaoctets/s). Cependant, leur endurance devra être la plus élevée possible (10^{12} écritures). À terme, le prix des mémoires SCM devrait converger vers celui des mémoires flash, permettant une utilisation massive dans les systèmes (voir figure 2.39). Nous regroupons dans le tableau 2.5 les principales caractéristiques de ces technologies.

Le développement de ces technologies n'est pas récent. On retrouve par exemple un article

Technologies	DDR	NAND	3DXP PCM	MRAM	STT-RAM	FeRAM	RRAM	NRAM
Persistance	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui
Endurance	10^{15}	10^3-10^5	10^6	10^{12}	10^{15}	10^{14}	10^6	10^{15}
Latence R/W	10ns/10ns	50us/25us	100ns/1us	50ns/1us	50ns/100ns	50ns/50ns	200ns/1us	10ns/10ns
Énergie (pJ)	0.005	1360	150	2	1	0.3	64	.005
Densité	moyenne	haute	moyenne	faible	faible	faible	moyenne	moyenne
Rétention	4us	1 an	10 ans	jours	jours	10 ans	10 ans	10 ans
Adressabilité	byte	Page	byte	byte	byte	byte	byte	byte
Scalabilité	16nm	QLC, 96L+	<20nm	28nm	28nm	40nm	<20nm	<10nm
Price /GB	7$	<1$	3.5$	2K$	4K$	32K$	3.5$	6$
Status	Prod.	Prod.	Prod.	Prod.	Prod.	Prod. / fin	échantillon	échantillon

Tableau 2.5 – État de l'art des différentes technologies SCM comparées à la DRAM et à la mémoire flash.

sur le développement de **mémoire PCM** (mémoire à changement de phase) daté de 1969 [Sie69]. C'est notamment la technologie utilisée pour les mémoires XPoint d'Intel [Han15]. Elle utilise certaines propriétés des matériaux chalcogénures tels que la photosensibilité ou la résistivité électrique. Ces matériaux subissent une transition de phase sous l'effet de la température. Cette transition de phase modifie leur conductivité ce qui permet d'encoder l'information. Le matériau peut être chauffé de deux façons : à l'aide d'un laser (utilisé pour les disques optiques réinscriptibles (CD-RW)) ou en utilisant l'effet joule. La lecture se fait ensuite en mesurant sa résistance à l'aide d'un courant suffisamment faible pour ne pas modifier son état. Ces mémoires, insensibles aux radiations sont particulièrement intéressantes dans des conditions d'utilisation inhabituelles celles de l'aérospatiale. **La MRAM** (*RAM magnétique*) utilise le spin des électrons. Suivant leur orientation par rapport à un aimant, la résistance change et permet de stocker l'information. En 2003, IBM a produit le premier démonstrateur avec la production de la première mémoire MRAM de 128kb [Bet+03]. Depuis, d'autres constructeurs ont produit de telles mémoires comme Freescale (un million de puces vendues), Samsung et Hynix. Le futur de cette technologie est d'évoluer en STT-RAM [Alv10] (*spin transfer torque* MRAM) permettant d'atteindre des densités d'intégrations plus élevées, une latence réduite proche de celle de la DRAM. Grâce à une consommation électrique faible, cette mémoire pourrait être adaptée pour les objets connectés (IOT). **La FeRAM** ou Mémoire Ferroélectrique à accès aléatoires possède la même structure que la DRAM avec un matériau ferroélectrique à la place du diélectrique. Cette technologie permettrait de profiter de la simplicité de la DRAM avec une intégration plus élevée (mais inférieure à celle de la mémoire flash [Alv10]). Beaucoup de problèmes sont rencontrés dans leur fabrication, notamment la pollution du silicium par le PZT. **La NRAM** (Nano RAM) repose sur l'utilisation de nanotubes de carbone. Un réseau de tubes croisés est activé électrostatiquement à l'aide d'une différence de potentiel pour faire fléchir les tubes et les mettre en contacte ou non [Ric08]. Ses performances proches de celles de la DRAM en font un excellent remplaçant. La production de mémoires utilisant cette technologie est relative-

ment simple et plusieurs couches peuvent être empilées pour augmenter la densité [Ger19]. **La ReRAM** (RAM résistive) se base sur le déplacement de trous dans des cristaux dopés. La production de cette mémoire est relativement simple et permettrait d'obtenir des prix compétitifs. Le désavantage de cette mémoire est la faible endurance aux écritures proche de celui de la flash. Le développement de cette technologie n'est pas encore terminé, et n'aboutirait pas avant 2025.

2.3.3 Nouvelles technologies d'interconnexion.

Dans la section précédente, nous avons discuté du besoin de développer de nouvelles technologies mémoire. Cependant, pour pouvoir profiter de débits plus élevés que ceux atteints actuellement, il est indispensable de développer de nouvelles technologies pour améliorer la performance du bus mémoire et du système d'interconnexion. Dans cette section, nous étudions les débits du bus mémoire et la consommation énergétique du système d'interconnexion des plateformes actuelles. Nous présentons ensuite de nouvelles technologies pouvant être utilisées.

2.3.3.1 Débit mémoire des processeurs

Nous proposons d'étudier l'utilisation d'un processeur Intel Skylake pour l'exécution d'applications HPC typiques. Un tel processeur possédant 28 coeurs cadencés à 2.3 GHz, réalise en performance crête 2 TFLOPS (voir détail du calcul dans la section 4.2). En supposant l'utilisation d'un bus mémoire délivrant 100 Gb/s, nous calculons le débit de donnée transférable pour chaque opération exécutée :

$$\frac{100 \times 10^9 \; byte/s}{2 \times 10^{12} \; FLOP/s} = 0.05 \; byte/FLOP \tag{2.8}$$

Ce résultat signifie que pendant l'exécution d'une opération, le système mémoire est capable de transférer 0.05 byte de la mémoire au processeur. Les applications de simulation numérique, telles que celles utilisées dans le domaine de la recherche pétrolière basée sur des algorithmes de Stencil, ont besoin de transférer depuis la mémoire 2 bytes par FLOP (4 pour l'application HPCG [29]), l'idéal se trouvant autour des 8 bytes par opération [Ber15]. Cette valeur est estimée pour un algorithme utilisant parfaitement la localité des mémoires cache et utilisant des opérations en double précision. On remarque donc que le système mémoire est très loin d'être capable de délivrer suffisamment de données pour permettre au processeur d'atteindre sa puissance crête (facteur 40).

29. Report on the HPC application bottlenecks - http://exanode.eu/wp-content/uploads/2017/04/D2.5.pdf

2.3.3.2 Consommation du système d'interconnexion

Les applications de calculs parallèles doivent échanger certains résultats entre serveurs. Il est courant d'utiliser des valeurs entre 0.1 et 0.2 $byte/FLOP$ [Ber15] pour des applications de simulation numérique. Pour pouvoir fournir suffisamment de données aux serveurs, la plateforme exascale devrait utiliser un système d'interconnexion avec un débit équivalent à :

$$0.2\ byte/FLOP \times 10^{18}\ FLOP/s = 200 \times 10^{15}\ byte/s \tag{2.9}$$

Les plateformes du Top500 allouent autour de 15% de leur budget énergétique total [Kog+08], pour l'alimentation du système d'interconnexion. Pour une plateforme Exascale consommant 20 MW cela correspond à une enveloppe énergétique de 3 MW. Nous pouvons ainsi calculer le budget d'énergie utilisable pour le transfert de chaque donnée :

$$\frac{3 * 10^6\ joule/s}{200 * 10^{15}\ byte/s} = 15\ pJ/byte = 1.87\ pJ/bit \tag{2.10}$$

Cette énergie représente l'enveloppe énergétique totale utilisable pour transférer un bit d'informations entre deux noeuds du système. Ce transfert comprend le passage dans les différents commutateurs. Or, comme nous l'avons expliqué dans la section 2.2.4.2, le budget actuellement nécessaire pour un tel transfert dépassait 1000 pJ/bit.

2.3.3.3 Objectifs des nouvelles technologies d'interconnexion

Nous constatons donc le besoin de nouvelles technologies d'interconnexion. La diminution de la consommation énergétique de trois ordres de grandeur ne pourra pas provenir d'une simple évolution des technologies actuellement utilisées :

— **La consommation électrique** doit être réduite de plusieurs facteurs, non seulement pour le système d'interconnexion, mais aussi pour les accès mémoires. La programmation d'algorithmes efficaces (localité) est alors primordiale.
— **Les débits mémoires** offerts doivent être améliorés de plusieurs facteurs pour pouvoir alimenter les processeurs et les accélérateurs.
— **La latence** du système d'interconnexion devra être la plus faible possible pour permettre à certaines applications de générer des accès mémoire imprédictibles (parcours de graphes). Les jeux de données ne pouvant pas être stockés sur une seule machine, le système d'interconnexion doit posséder une latence très faible pour pouvoir accéder rapidement à une donnée distante.

Afin de répondre aux défis exposés précédemment, l'industrie du HPC s'est lancée depuis plusieurs dizaines d'années dans la recherche et le développement de technologies photoniques.

2.3.3.4 La photonique

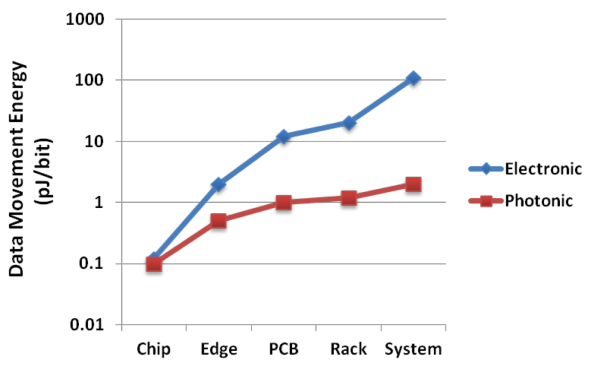
La photonique est un domaine de la physique qui a pour objectif de manipuler la lumière (photon) : l'émission, la transmission, la captation et le traitement. Les premiers émetteurs ont été développés dans les années 1960. Aujourd'hui, la photonique a de nombreuses applications : lecteur de code-barre et DVD, illumination LED, vision nocturne... Dans le domaine des télécommunications, l'application la plus connue est la fibre optique. L'utilisation de technologies photoniques dans les supercalculateurs a deux avantages principaux répondant aux besoins évoqués ci-dessus.

— **L'indépendance à la distance** est le premier avantage de la photonique. Que ce soit en termes de latence ou de consommation électrique, la photonique permet de *réduire les distances* d'un supercalculateur. En effet, la génération d'un signal optique ou électronique nécessite une quantité d'énergie similaire. Cependant, pour les technologies photoniques, celle-ci évolue faiblement avec la distance de communication (figure 2.40a). Lorsque les accès sont proches (sur la puce), il n'y a pas d'avantages en terme énergétique à utiliser la technologie photonique. Cependant, une fois les photons générés, le coût énergétique ne varie que faiblement avec la distance d'accès, et cela jusqu'à plusieurs centaines de mètres. Ainsi, le coût énergétique d'un accès mémoire est proche de celui d'un accès à une mémoire distante située sur un autre serveur. Pour ces accès, la photonique permet de réduire la consommation électrique de plusieurs ordres de grandeur.

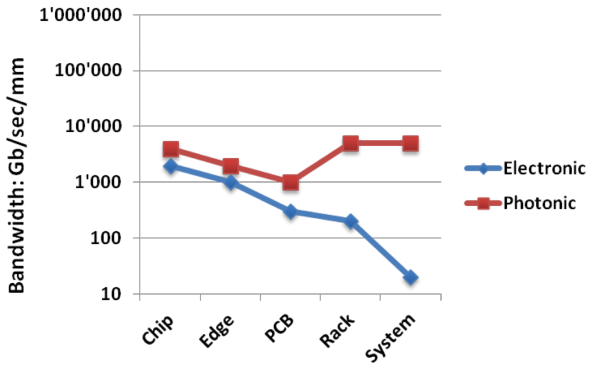
— **Les débits mémoire**s atteignables par l'utilisation de la photonique sont le deuxième avantage. À la différence des signaux électriques qui interfèrent lorsqu'ils sont utilisés sur une même broche, il est possible d'accumuler plusieurs signaux de différentes longueurs d'onde sur le même guide d'onde (*wave guide*). Il est ainsi possible d'atteindre une haute densité de communication (voir figure 2.40b). Ces technologies peuvent atteindre des débits de 800 Gb/s avec une consommation de 2.2 pJ/bit. Les prochaines générations permettront d'atteindre des débits de 1 Tb/s avec une consommation de 1 pJ/bit [Ber18]. Cette faible densité va permettre d'utiliser la photonique à l'intérieur des puces permettant d'atteindre des débits élevés pour par exemple connecter une mémoire HBM au processeur.

L'utilisation des technologies photoniques dans les supercalculateurs va permettre de *réduire* les distances en améliorant les latences de communication ainsi qu'en réduisant les coûts énergétiques. Il sera donc presque équivalent de communiquer avec sa propre mémoire, qu'avec celle d'un serveur situé à l'autre bout du centre de données. Ainsi, les supercalculateurs utilisés par plusieurs utilisateurs subiront moins les effets de fragmentation, car il n'y aura plus d'inconvénient à utiliser des machines distantes. Cependant, la topologie des calculateurs va devoir être repensée pour exploiter cette caractéristique (voir section 2.3.5).

De nombreuses technologies sont développées pour rendre possible l'utilisation de la photonique dans les centres de données. Par exemple, en utilisant des anneaux résonnants directement sur les puces silicium (voir figure 2.41). Ces anneaux sont des guides d'onde bouclés, permettant, pour une longueur d'onde donnée, de générer une résonance et propager l'onde dans celui-ci. Pour pouvoir contrôler les signaux, différents types d'anneaux ont été dévelop-

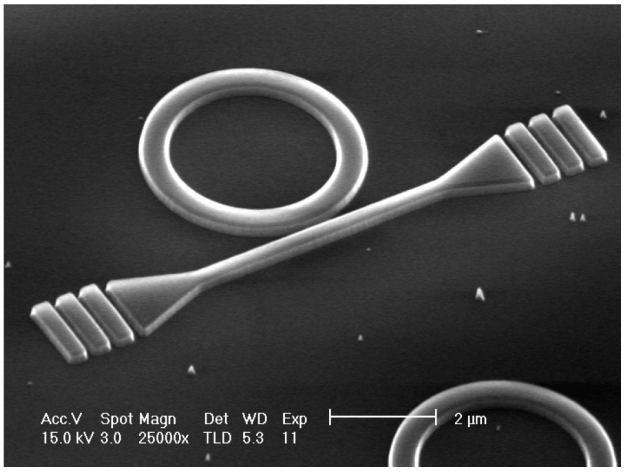


(a) Distance d'accès et consommation électrique.

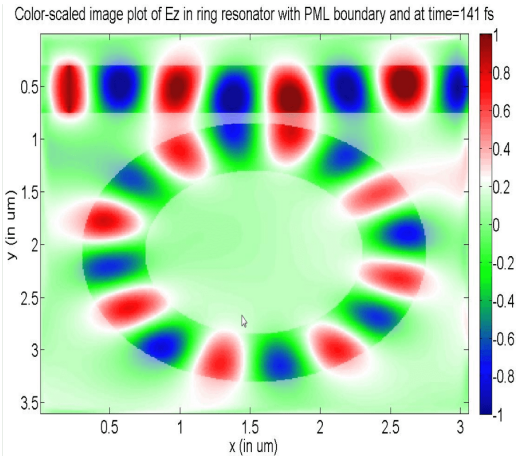


(b) Distance d'accès et débit mémoire.

FIGURE 2.40 – La transmission par photonique a deux avantages principaux comparés à la transmission électronique[Luc+14] : diminution de la consommation électrique (a) et amélioration du débit (b).



(a) Anneau résonnant vu au microscope.



(b) Fonctionnement d'un anneau résonnant

FIGURE 2.41 – Illustration d'un anneau résonnant

pés (voir figure 2.42). Chaque anneau a une fonctionnalité permettant de réaliser différentes opérations : laisser passer ou non un signal, le commuter ou encore détecter la présence d'une certaine longueur d'onde. D'autres types d'anneaux sont utilisés pour générer les signaux et permettent d'encoder les informations à transmettre. En combinant ces anneaux, il est possible de développer des composants complexes comme des commutateurs.

2.3.4 Nouvelles architectures

Dans cette section nous présentons les nouvelles architectures et les méthodes employées pour améliorer les performances des processeurs et des accélérateurs utilisés dans les super-

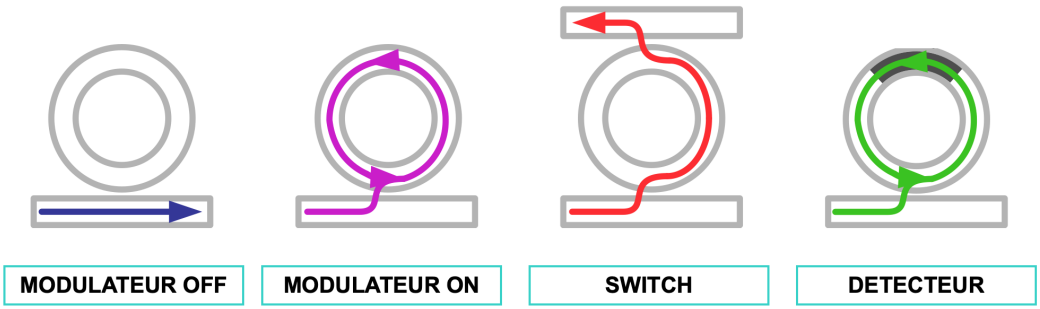


FIGURE 2.42 – Différents types de résonateur en anneau.

calculateurs. Pour cela nous présentons la méthode de *co-design* et présentons quelques architectures novatrices. Enfin, nous discutons de la nécessité d'avoir des architectures hétérogènes pour atteindre l'efficience énergétique espérée.

2.3.4.1 Codesign

En informatique, le co-design (ou co-conception) est un processus de conception de système informatique impliquant différents acteurs : fabricant, programmeurs, utilisateurs finaux. Ce processus a pour objectif d'influencer la conception des architectures et le développement de technologies en tenant compte des besoins logiciels et algorithmiques. Il regroupe autour de partenariat longue durée l'expertise des fournisseurs, des architectes, des scientifiques et des programmeurs dans le but de développer les meilleures technologies possibles en fonction des besoins exprimés (coût, performance, efficacité énergétique). Le co-design est une piste majeure pour le développement de processeurs et d'accélérateurs ultras optimisés utilisés dans l'élaboration de plateformes exascale. La motivation principale étant de réfléchir à la façon de développer une plateforme exascale répondant aux besoins des applications plutôt que de se demander quelles applications sont adaptées à cette plateforme une fois celle-ci développée [PT13].

Le développement en co-design suit un processus d'aller-retour entre la partie logicielle et la partie matérielle (voir figure 2.43). Les utilisateurs de HPC (programmeurs, scientifiques) expriment les besoins de leurs applications aux constructeurs de matériels : débit mémoire, latence, scalabilité, puissance de calcul. Cette étape peut être difficile, car certaines applications comptent plusieurs millions de lignes de codes, et l'expression de leurs besoins est loin d'être triviale. Il peut alors être intéressant d'utiliser des applications plus simples se comportant comme les applications réelles (benchmarks, micro-benchmarks). En étant conscients des besoins des applications, les fabricants peuvent développer des architectures adaptées. Pour cela, il est nécessaire d'utiliser toutes les avancées technologiques réalisées dans différents domaines (mémoire, interconnexion). Ensuite, les constructeurs peuvent communiquer aux développeurs certaines spécificités du matériel qui peuvent être exploitées par le logiciel (caractéristiques de la hiérarchie mémoire, instructions utilisables). Le principal frein de cette méthode est économique. Il faut que l'utilisation d'une architecture puisse bénéficier au maximum d'applications

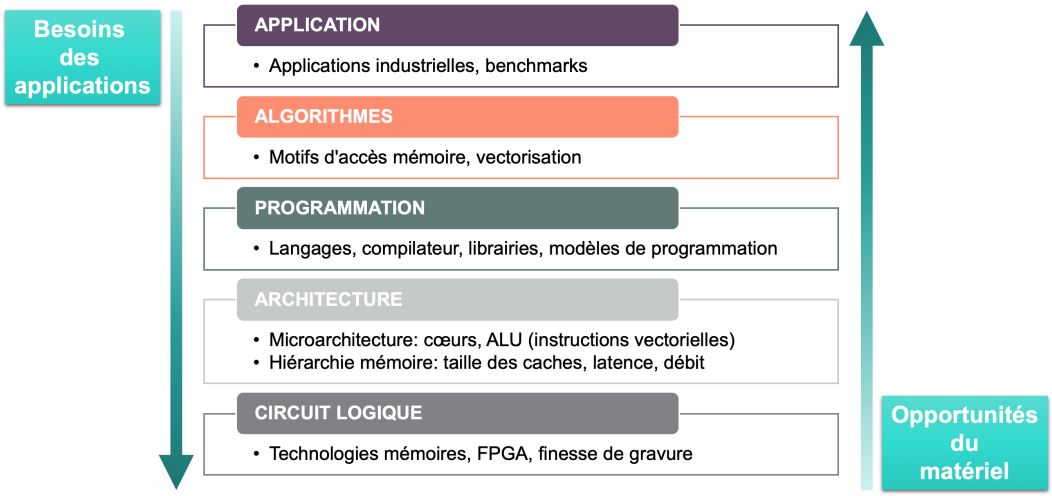


FIGURE 2.43 – Processus de co-design.

pour qu'un constructeur prenne la décision de la développer.

Ces solutions pourront être développées pour accélérer certains algorithmes ou motifs de calcul [Asa+06] : algèbre linéaire (matrices creuses ou denses), méthode spectrale (transformation de Fourier rapide), algorithmes de Monte-Carlo, graphe, programmation dynamique. Un exemple de co-design vise à adapter les unités de calcul aux besoins des applications (instructions vectorielles, précision). Lorsque celles-ci ne nécessitent pas de réaliser des calculs avec une grande précision, il peut alors être intéressant d'utiliser des registres plus petits. L'étude [Hor14] montre que les opérations flottantes sur des registres plus petits consomment moins d'énergie : 0.03 pJ pour une addition sur 8 bits contre 0.9 pJ pour une addition sur 32 bits. De plus, le coût énergétique d'une multiplication évolue avec la taille (n) des registres utilisés ($O(n^2)$) ainsi que la latence ($O(n)$) [Sze+17]. Enfin, le nombre de transistors nécessaires est réduit avec la taille des registres ($36\mu m^2$ contre $4184\mu m^2$) pour des additionneurs de 8 et 32 bits [Hor14]. Cette technique a été utilisée par Google pour le développement de son ASIC en 2015. Le TPU est un ASIC optimisé pour réaliser la phase d'inférence des modèles d'apprentissage. Cette phase ne nécessite pas d'avoir autant de précision que la phase d'entraînement. Les ingénieurs ont donc développé le TPU en utilisant 65536 registres de 8 bits. Cette réduction d'un facteur 8 permet de réduire la consommation électrique et la taille des circuits par un facteur 6 [Jou+17]. La latence de réponse, importante pour la phase d'inférence, est réduite entre 15 et 30 fois par rapport au GPU équivalent de l'époque (Nvidia K80).

2.3.4.2 Quelques nouvelles architectures

Pour répondre aux défis exposés dans la section précédente, de nombreuses entreprises se lancent dans le développement de nouvelles architectures, qui peuvent être très différentes de celles que nous utilisons depuis 30 ans. En janvier 2018, une étude [Met18] a dénombré 45 start-ups qui développaient des circuits spécialisés pour certaines applications d'intelligence ar-

	Coeurs/Thread	Freq. (GHz)	Débit mémoire (GB/s)	Perf. DP (TFLOPS)	TDP (Watt)	Eff. (GFLOPS/watt)
Intel Xeon 6246	12 / 24	3.30	140 (DRAM)	1	165	6
Nvidia V100	5120	1.5	900 (HBM2)	7.8	300	26
Pezy-SC2.2	2048 / 16384	1	2000 (DRAM + HBM2)	4.1	180	23
Fujitsu A64FX	48	Non communiquée	1000 (HBM2)	2.7	180	15

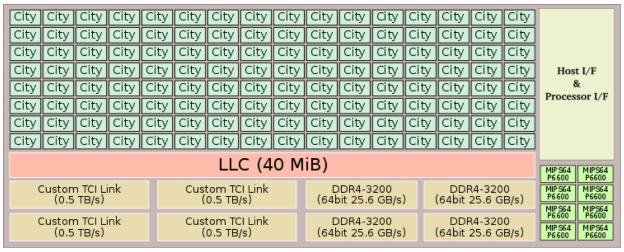
Tableau 2.6 – Caractéristiques et performances d'architectures utilisées dans les supercalculateurs les plus efficaces du Top500.

tificielle : analyse de voix, conduite autonome... Cinq d'entre elles avaient alors levé plus de 100 millions de dollars d'investissement. Les derniers classements du Top500 ont vu l'apparition de supercalculateurs utilisant de nouveaux accélérateurs ou l'évolution d'anciennes architectures. Par exemple, les GPU Nvidia utilisaient seulement de la mémoire GDDR5 sur les cartes des architectures Keppler. La génération suivante (Volta) a permis d'utiliser un interposeur en silicium connectant directement au processeur une mémoire HBM2. Ceci a permis de réduire les coûts énergétiques associés aux communications de 100 pJ/bit à moins de 10 pJ/bit. L'efficacité énergétique des supercalculateurs a ainsi pu être améliorée, passante de 6,7 à 14,1 GFLOPS/-watt. En plus de nouvelles cartes GPU (Nvidia V100), deux nouveaux accélérateurs sont utilisés dans les supercalculateurs classés en tête du Green500 : le PEZY-SC2 et le processeur ARM A64FX. Nous comparons les principales caractéristiques de ces architectures avec celle d'un processeur Intel moderne (génération Kaby Lake) dans le tableau 2.6.

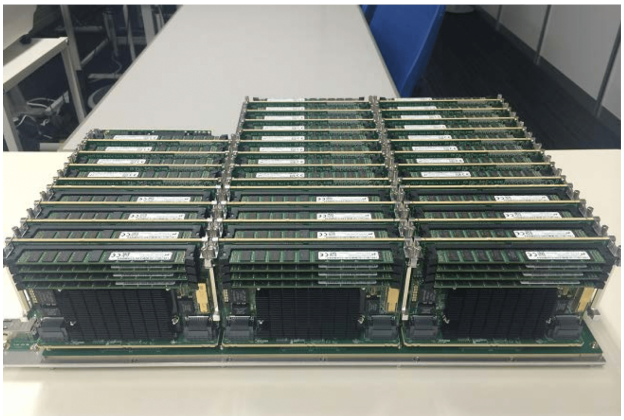
Les architectures PEZY-SCx sont des accélérateurs multicoeurs développés par la société PEZY. Ils sont utilisés dans plusieurs supercalculateurs (ZettaScaler) construits au Japon. La dernière version, PEZY-SC2, a une surface six fois plus grande que le processeur Intel auquel il est généralement associé. La puce dispose de quatre canaux mémoires permettant d'atteindre une bande passante de 95.37 GB/s. En plus, l'accélérateur utilise une technologie sans fil (ThruChip Interface (TCI)) pour communiquer avec une mémoire 3D (voir figure 2.44a). Chaque interface peut atteindre 500 GB/s pour un total de 2 TB/s.

Le processeur ARM A64FX développé par Fujitsu est le processeur équipant le premier supercalculateur classé au Green500. Il possède 48 coeurs capables d'exécuter des instructions vectorielles de 512 bits. Il est doté de 32 GB de mémoire HBM2 permettant d'atteindre la performance de 2.7 TFLOPS (10^{12} FLOPS). Alors que le processeur A64FX n'est pas le plus efficace des 4 processeurs listés dans le tableau 2.6, il est pourtant utilisé dans le premier supercalculateur du Green500. En effet, nous remarquons que la puissance électrique d'une architecture ne suffit pas à déterminer l'efficacité énergétique d'une solution complète. Le reste de la plateforme doit lui aussi rentrer en compte (mémoire, utilisation du port PCI-E). La densité d'installation des architectures peut aussi avoir un impact (voir figure 2.44b). Ainsi, bien que le GPU Nvidia V100 soit le plus efficace, ce sont les solutions utilisant les processeurs ARM et PEZZY qui sont les mieux classées.

29. Comparaison des classements du Top500 entre juin 2016 et 2017 des supercalculateurs Shoubu et Tsubame3.0



(a) microarchitecture de l'accélérateur.



(b) Plusieurs PEZY-SC2 peuvent être connectés sur un serveur.

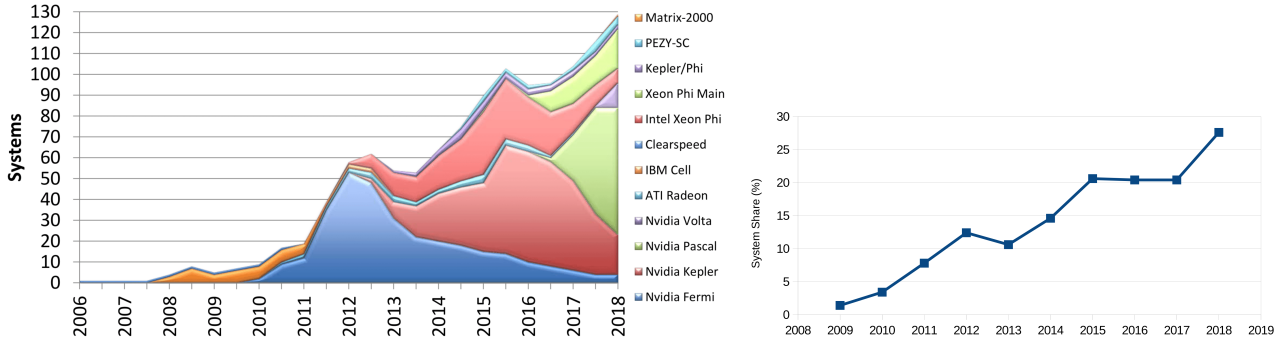
FIGURE 2.44 – Accélérateur PEZY-SC2 [Ryu18]

2.3.4.3 Hétérogénéité

Avec la fin de la validité de la loi de Dennard et les premiers signes d'apparition du "mur" de l'énergie, les constructeurs ont été contraints de repenser la microarchitecture des processeurs. Les processeurs utilisés étaient alors désignés comme d'usage général (*general purpose processor* (GPP)). Permettant d'exécuter tout type d'application, ces processeurs ne sont optimisés pour aucune d'entre elles. La performance et l'efficacité énergétique sont les deux raisons qui ont poussé la communauté HPC à adopter cette technologie. L'évolution des langages et des libraires a permis de faciliter les développements d'application pour ces accélérateurs. En 2007, Nvidia publiait CUDA, un langage propriétaire permettant de programmer les cartes graphiques de la marque ainsi que des librairies de calculs mathématiques optimisées (CUBLAS, CUFFT). Ainsi, le nombre de supercalculateurs hétérogènes présents dans le Top500 n'a fait qu'augmenter ces dix dernières années pour atteindre près d'un supercalculateur sur trois en 2019 (voir figure 2.45b). À partir de 2010, les premiers clusters contenant des cartes graphiques apparaissent au classement du Top500 (voir le graphique 2.45a). Il faut attendre 2011 pour voir une réelle percée de cette technologie. En 2018, les GPUs étaient les principaux accélérateurs utilisés dans les supercalculateurs (93% des plateformes hétérogènes du Top500 en 2018).

En 2020, la majorité des supercalculateurs n'utilise cependant pas d'accélérateurs. En effet, les plateformes homogènes ont plusieurs avantages. La performance des processeurs GPP peut être suffisante pour une majorité d'applications. L'utilisation d'une seule famille de processeur facilite aussi la gestion du centre de calculs. Aussi, l'utilisation d'un processeur type x86 assure une stabilité à long terme, où l'utilisation de la génération suivante d'une architecture est similaire et ne nécessite généralement pas de grosse transformation de code.

D'un autre côté, utiliser des architectures hétérogènes peut présenter plusieurs difficultés. La première vient de la nécessité de transformer ou de reprogrammer les applications pouvant nécessiter l'utilisation de langages ou des modèles de programmation différents. Ces transforma-



(a) Évolution des différents types d'accélérateurs utilisés [Str18].

(b) Évolution du pourcentage de supercalculateurs utilisant un accélérateur [Kha19].

FIGURE 2.45 – Évolution de l'utilisation d'accélérateurs dans les supercalculateurs.

tions demandent un gros investissement des programmeurs. D'autres difficultés concernent la gestion et l'utilisation des ressources. Les supercalculateurs sont utilisés pour exécuter différents types d'applications, avec différents besoins. Utiliser un seul type d'accélérateur ne sera pas efficace pour toutes les applications. Le nombre de combinaisons engendrées par la disponibilité de plusieurs types d'accélérateurs, de capacité de mémoire différente complique l'obtention des performances optimales. La recherche des configurations optimales (compilateur, drapeaux de compilation, type d'accélérateur, taille de découpage des jeux de données, nombre de coeurs et fréquence utilisés) est alors difficile à réaliser manuellement. Pour explorer les différentes configurations, il peut alors être intéressant de se tourner vers des techniques d'autoréglage (*auto-tuning*) [Dat+08 ; HE08 ; MTB11 ; Cas+15 ; Pop16 ; Ben+14a]

Une récente étude [And+19] conduite auprès de plusieurs industries montre que l'investissement nécessaire pour réaliser ces transformations est un frein majeur à l'adoption de ces architectures. La transformation du code est loin d'être évidente et varie en fonction des accélérateurs choisis (voir figure 2.46). Les langages, librairies et outils de programmation (déboguer, suivi de performance) sont moins avancés et moins robustes que ceux utilisés sur des architectures classiques. Les quatre entreprises interviewées [And+19] constatent le manque d'outils adaptés pour réaliser le travail du portage de code et de validation de performances. C'est une différence majeure entre l'industrie et le domaine de la recherche. Les applications industrielles sont plus complexes que celles utilisées comme démonstrateurs et il est souvent plus difficile d'atteindre les performances théoriques. Une entreprise témoignant dans l'étude [And+19] affirme que le manque d'expertise était le principal défi pour le portage d'application. Ce constat est partagé par les autres entreprises présentées comme grandes (*large*) et ayant les moyens d'embaucher de potentiels experts.

Cependant, les pressions énergétiques et économiques obligeront les plus grands centres à se doter de plateformes hétérogènes malgré les difficultés associées.

Dans le projet exascale, l'hétérogénéité ne viendra pas seulement de l'utilisation d'accéléra-

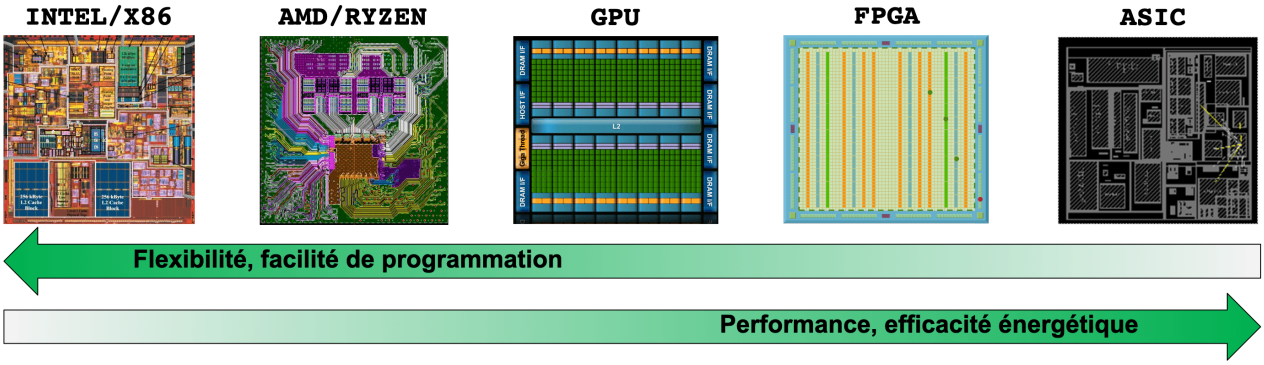


FIGURE 2.46 – Différentes technologies sont disponibles, chacune ayant ses avantages et inconvénients.

teurs différents. Les architectures elles-mêmes seront développées à partir de différentes technologies, pour maximiser l'efficacité énergétique. Le co-design associé aux technologies photoniques va permettre le développement d'architectures ultra-optimisées permettant la construction d'une plateforme exascale consommant entre 20 et 30 MW. Les puces des processeurs vont être adaptées à l'application pour minimiser les échanges de données (voir figure 2.47). Un même processeur pourra alors disposer de différents types de coeurs (complexes (x86), simples (GPU), programmables (FPGA)) ayant des mémoires aux caractéristiques, elles aussi différentes (capacité, latence, débit, politique de remplacement, associativité). Par exemple, une application d'apprentissage automatique peut avoir besoin pendant la phase d'entraînement de ressources similaires aux GPU. Dans une seconde phase (inférence), un autre module (FPGA, ASIC), directement accessible sur la puce, peut alors être utilisé. Ainsi, les données ne nécessitent par d'être transférées sur un autre serveur et peuvent être encore présentes dans la hiérarchie mémoire (mémoire 3D, SCM, DRAM).

2.3.5 Gen-Z

Cette section est consacrée à la présentation d'un nouveau protocole de communication nommé Gen-Z. Pour cela, nous résumons les principales motivations de la nécessité d'utiliser une telle technologie. Nous présentons ensuite les principales caractéristiques et les avantages du protocole. Enfin, nous discutons de l'opportunité apportée par Gen-Z pour repenser fondamentalement l'architecture de nos plateformes.

2.3.5.1 Motivations

Le tsunami de données générées présenté dans la section 2.2.4 nécessite de repenser la façon dont nous les traitons. Actuellement ces données sont envoyées aux centres de calculs pour être analysées. Le volume de données générées dans les prochaines années nous empêchera de poursuivre cette méthode. Une voiture connectée génère par exemple entre 2 et 5 terabytes de données chaque jour. Le coût des technologies et des infrastructures nécessaires pour les

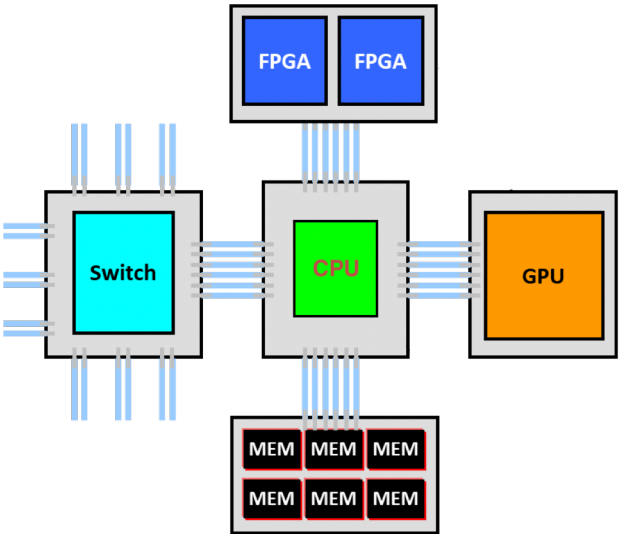


FIGURE 2.47 – Développement de processeurs très hétérogènes possédant différentes technologies sur la même puce et communiquant par photonique [Ber18].

transférer vers les centres de données serait alors trop élevé. De plus, la conduite autonome comme d'autres applications (villes connectées) nécessite d'avoir des réponses rapides. La seule solution viable est alors de les traiter le plus proche possible de leur zone de création. Seule une partie d'entre elles sera alors remontée aux centres de calculs pour l'archivage ou améliorer l'apprentissage des réseaux de neurones.

Les voitures et les objets connectés produisent des données sensibles, et la nécessité d'utiliser des plateformes de traitement sécurisées est primordiale. Chaque composant des solutions est une source d'attaque potentielle : capteurs, processeurs, mémoires, réseaux. Les dégâts potentiels d'une attaque sur ces sites ultra-connectés pourraient alors être catastrophiques (attaque de la signalisation routière, de voitures connectées ou d'un système de refroidissement d'une centrale nucléaire).

Concernant le domaine du HPC, nous avons expliqué dans les sections précédentes que de nombreuses technologies (mémoires, processeurs, accélérateurs) étaient en développement. Ces nouvelles architectures sont produites par différents constructeurs. De plus, les architectures actuelles ont été développées pour un nombre limité de technologies (mémoire DRAM, processeur x86 ou ARM, extension PCIe). Ainsi, la principale difficulté pour leur utilisation viendra de leur inter compatibilité.

2.3.5.2 Limites des architectures actuelles

Malgré l'évolution des accélérateurs et l'utilisation de nouvelles technologies mémoire, les architectures actuelles ne pourront pas évoluer indéfiniment et ont déjà montré leurs limites. La figure 2.48, représente l'évolution des différentes caractéristiques liées aux bus de communication d'un processeur ainsi que l'évolution du nombre de coeurs disponibles. Nous constatons que l'évolution du nombre de coeurs a évolué plus rapidement que celle des performances du système

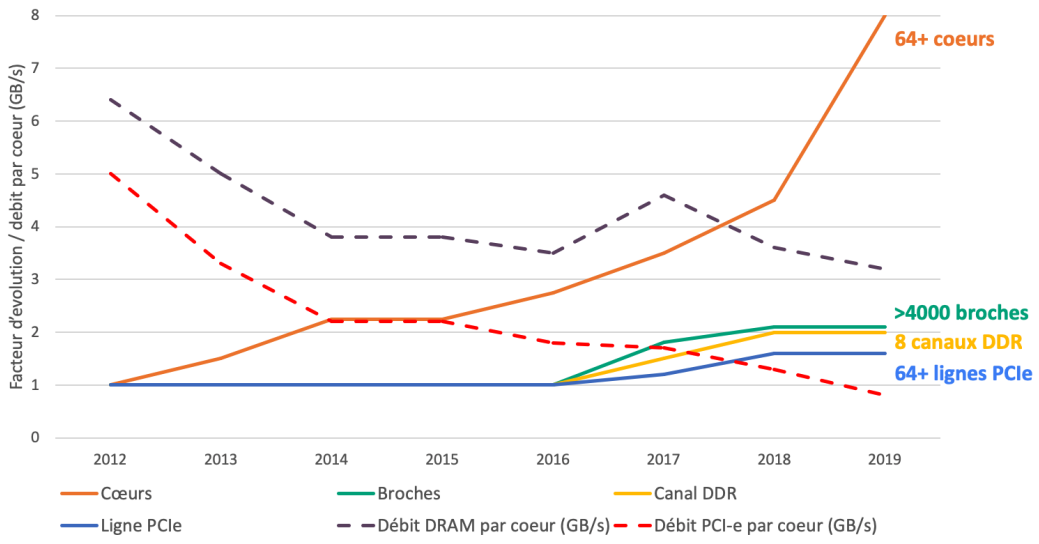


FIGURE 2.48 – Évolution annuelle des caractéristiques principales des architectures (nombre de coeur, débit mémoire) et leur impact sur les débits disponibles par coeur.

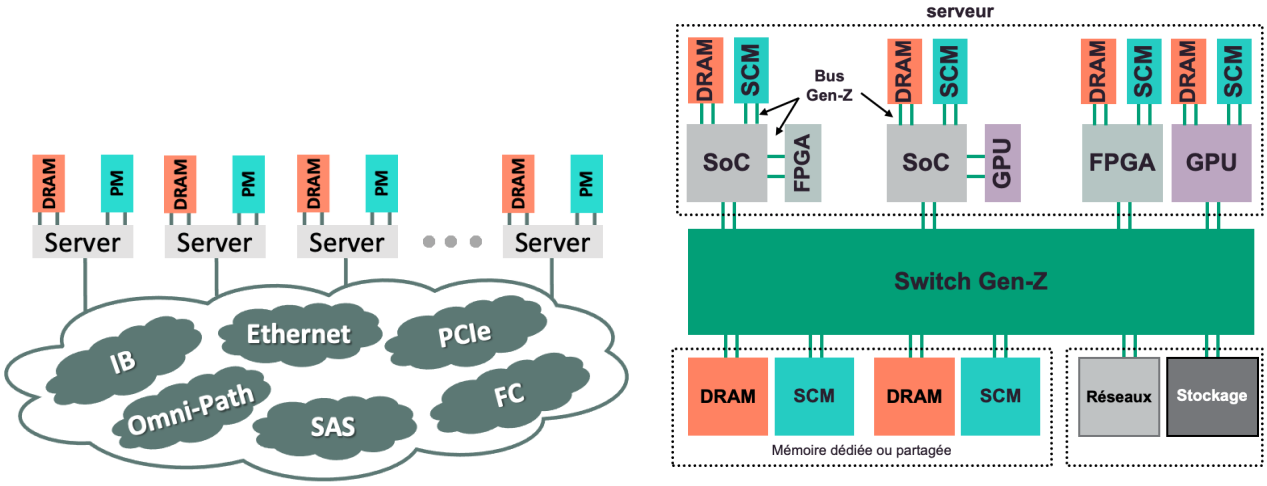
mémoire. En effet, lorsque le nombre de canaux mémoires a été multiplié par 2 (passant de 4 à 6, puis à 8 canaux) le nombre de coeurs a été multiplié par un facteur 8. Le débit du bus mémoire (courbe violette) et du bus PCIe (courbe rouge) disponible par coeur a donc diminué. La majorité des broches du processeur est déjà utilisée par les canaux mémoires et il sera très difficile d'en allouer plus.

De plus, le manque d'espace disponible sur la carte mère nous oblige à développer une solution permettant d'augmenter les débits de plusieurs facteurs avec les canaux actuellement utilisés. L'espace disponible autour du processeur pour ajouter des emplacements mémoires est aussi de plus en plus limité. Les plus grosses configurations actuelles peuvent atteindre jusqu'à 1.5 TB de mémoire, insuffisant pour traiter les jeux de données envisagés.

Avec le modèle actuel, l'évolution des mémoires ou des processeurs est verrouillée. La nécessité que l'un soit compatible avec l'autre oblige les deux parties à évoluer de manière synchronisée. Par exemple, avant de pouvoir utiliser des mémoires DDR5, il faut que les processeurs les supportant soient développés. Cette dépendance est un frein à l'évolution.

Le développement des mémoires SCM permettra à terme d'obtenir des performances proches de la DRAM. Cependant, elles ne permettront pas d'augmenter les débits mémoires de plusieurs facteurs. En effet, ces mémoires sont installées sous forme de barrettes mémoire (NVDIMM voir section 2.3.2.3) ou sous forme d'extensions de carte PCI (comme les mémoires Intel Optane). Que ce soit sous forme de barrette ou de carte PCI, la restriction de performance vient du bus utilisé. De plus, l'utilisation d'extension PCI a ses limites, car ce bus ne supporte pas la cohérence de cache des mémoires associées (contrairement au bus mémoire).

Un autre défaut des architectures actuelles impactant la performance des applications concerne la multiplicité des protocoles utilisés : DDR, PCIe, infiniband, Ethernet, sata (voir figure 2.49a). L'utilisation de différents protocoles à un coût, impactant la latence, le débit et



(a) De nombreux protocoles sont impliqués lors d'une communication.

(b) Le protocole Gen-Z permet d'interconnecter tous les composants d'une plateforme.

FIGURE 2.49 – Gen-Z permet d'utiliser un unique protocole pour la totalité des communications.

l'énergie consommée pour chaque transfert. Pour développer une plateforme exascale efficace, il est donc nécessaire de revoir l'utilisation de ces protocoles.

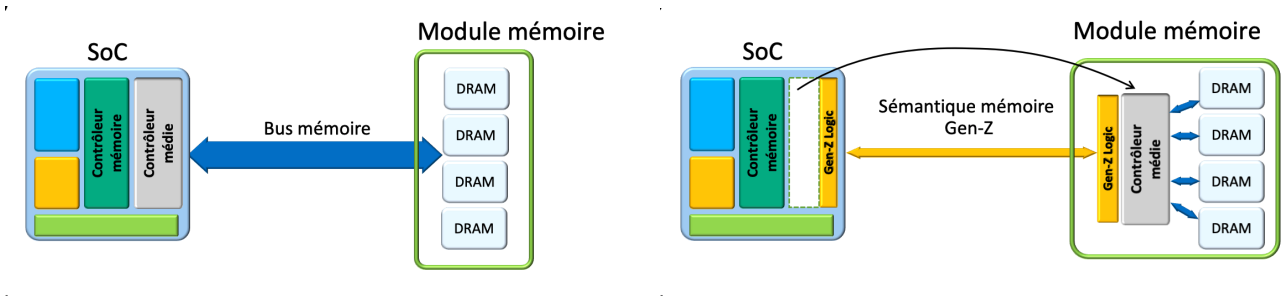
2.3.5.3 Gen-Z.

Gen-Z est un protocole universel d'interconnexion, de puce à puce, permettant les échanges entre composants informatiques au travers de communications à sémantique en mémoire. Gen-Z est dit universel, car il permet de connecter différentes architectures (CPU, GPU, FPGA) ainsi que différents médias (mémoire, stockage, archive) à travers un unique protocole (voir figure 2.49b). Ces communications (asynchrones) peuvent être locales (entre les composants d'un même serveur), ou bien externes (entre deux serveurs).

Lancé en 2016, il est développé par un consortium qui compte aujourd'hui plus de 70 membres[30], dont les plus grandes entreprises du domaine informatique : AMD, ARM, Cisco, Dell, Google, HP, HPE, Micron, Microsoft, Redhat, Samsung... Il est important de noter l'absence de deux constructeurs majeurs (Intel et Nvidia) qui ont rejoint un deuxième consortium pour le développement d'un second protocole nommé CXL[31]. Le développement de CXL est moins avancé que celui de Gen-Z.

Ce protocole a été développé pour répondre aux nombreux défis posés par l'utilisation des objets connectés et la nécessité de traiter des grands jeux de données. Gen-Z permet d'adresser un espace mémoire 2^{92} Byte (soit 4096 yottabytes, soit mille fois plus grands que notre espace numérique actuel) et d'interconnecter 16 millions d'objets. Ces objets pouvant être des compo-

30. Liste des membres du consortium Gen-Z - https://genzconsortium.org/about-us/membership
31. Compute Express Link - https://www.computeexpresslink.org/



(a) Actuellement les technologies mémoire et celles des processeurs sont liées.

(b) Avec Gen-Z, chaque module aura sa propre interface.

FIGURE 2.50 – Déverrouiller l'évolution technologique grâce à Gen-Z.

sants d'un serveur (mémoire, processeur, accélérateurs) ou des objets connectés plus complexes.

Sémantique mémoire. L'avantage de Gen-Z est d'utiliser des communications à sémantique mémoire. Tous les composants sont considérés comme des modules mémoires et peuvent être accédés grâce à des instructions *load/store*. Il est ainsi possible d'adresser les mémoires SCM par byte et non plus par bloc (héritage dû aux disques optiques). En réduisant la complexité des instructions, le processeur et le système d'exploitation ne sont pas impliqués dans leur exécution, libérant des ressources pour l'exécution d'autres instructions. Gen-Z propose d'autres instructions permettant de réaliser des opérations plus complexes telles que les opérations atomiques (comparaison, addition), des interruptions ou encore de gérer la cohérence des caches.

Comme exposé en présentation de cette section, les évolutions technologiques des différentes parties d'une architecture sont dépendantes les unes des autres. Gen-Z permet de supprimer ce verrou. La figure 2.50 présente comment les processeurs et les mémoires vont utiliser les interfaces Gen-Z pour être indépendantes. Le processeur et la mémoire auront une interface Gen-Z et l'évolution d'une partie ne nécessitera pas de changement de l'autre. De plus, en spécifiant un protocole universel, Gen-Z assure l'interopérabilité entre tout type de composant qui possède une interface (CPU, FPGA, GPU, DSP, I/O, mémoires).

Latence et débit. Le protocole Gen-Z permet d'améliorer les performances du système d'interconnexion et apporte aussi de nouvelles fonctionnalités permettant de repenser l'architecture des plateformes. La réduction du nombre de protocoles et des différentes couches impliquées dans le transfert des données (système de fichier, tampon I/O, drivers) permet de réduire les latences des accès. Les spécifications actuelles permettraient de réduire le nombre d'instructions de 25000 à 3 pour réaliser un transfert d'une donnée localisée sur un disque. En utilisant des technologies SCM, les latences d'accès entre un processeur et une mémoire située sur un autre serveur seraient alors proches de celles d'un accès à la mémoire DRAM locale (inférieur à 250 ns).

Gen-Z permet d'atteindre des débits mémoires supérieurs à ceux des bus mémoires actuels. En fonction des technologies et du nombre de broches utilisées, les débits mémoires d'une

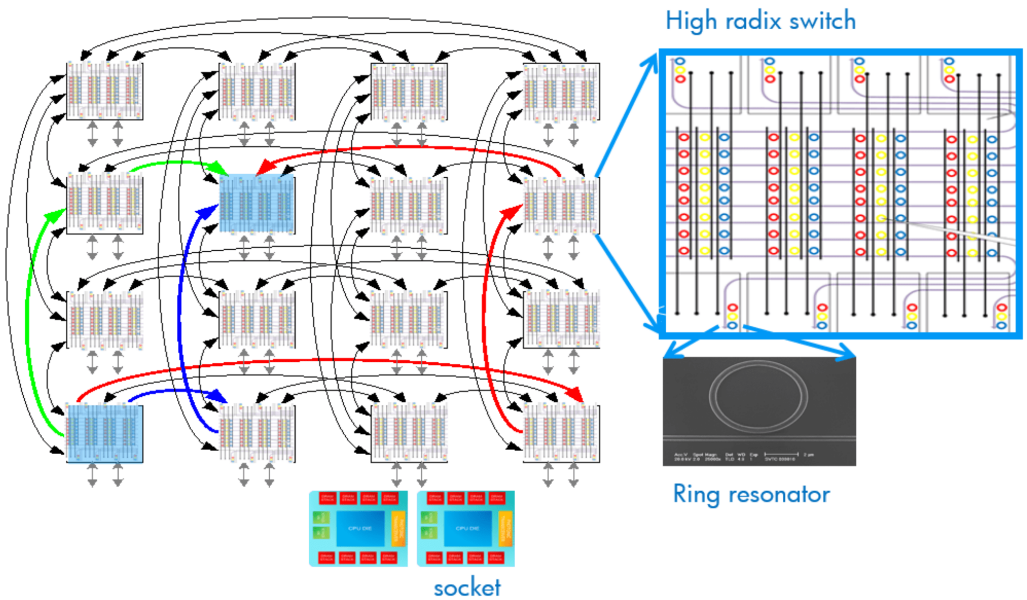


FIGURE 2.51 – La topologie HyperX permet de joindre n'importe quel serveur grâce à trois routes différentes.

architecture Gen-Z peuvent atteindre 1.7 TB/s. Ces performances sont possibles grâce au débit atteignable pour chaque broche du processeur utilisée. Il sera donc possible d'atteindre des débits mémoires 10 fois supérieurs avec seulement la moitié de broches. Ceci permettra de développer des processeurs moins coûteux.

Topologie Gen-Z supporte nativement différentes topologies (P2P, mesh, Torus) pour réaliser la connexion entre les composants, mais aussi entre les serveurs (voir figure 2.52). Ces topologies permettent d'implémenter des réseaux multi chemins (*multipath*) et de gérer le contrôle de congestion. Si un lien entre deux composants est cassé, le protocole s'adapte automatiquement en passant par d'autres chemins disponibles. Les différents composants peuvent alors servir de relais pour transmettre les paquets. Aujourd'hui lorsqu'un serveur tombe en panne, les données présentes dessus sont perdues, et les applications doivent être relancées. Gen-Z et les SCM permettront d'assurer une connexion permanente aux différents modules avec l'utilisation de la topologie HyperX [Ahn+09] (voir figure 2.51), qui permet de joindre n'importe quel serveur avec trois routes différentes. Cette redondance permet d'éviter les congestions et d'assurer une utilisation maximale du supercalculateur même en cas de panne d'une des connexions.

Sécurité La sécurité est un aspect fondamental du développement de Gen-Z. La sécurisation des échanges est en effet très importante si ce protocole doit être utilisé à l'extérieur des centres de calculs. Chaque composant Gen-Z est identifié lors de leur création en usine. Il est ainsi possible d'authentifier n'importe quelle transaction. Le protocole peut ainsi permettre à un routeur de bloquer ou non une communication. La sécurité est réalisée par un logiciel centralisé

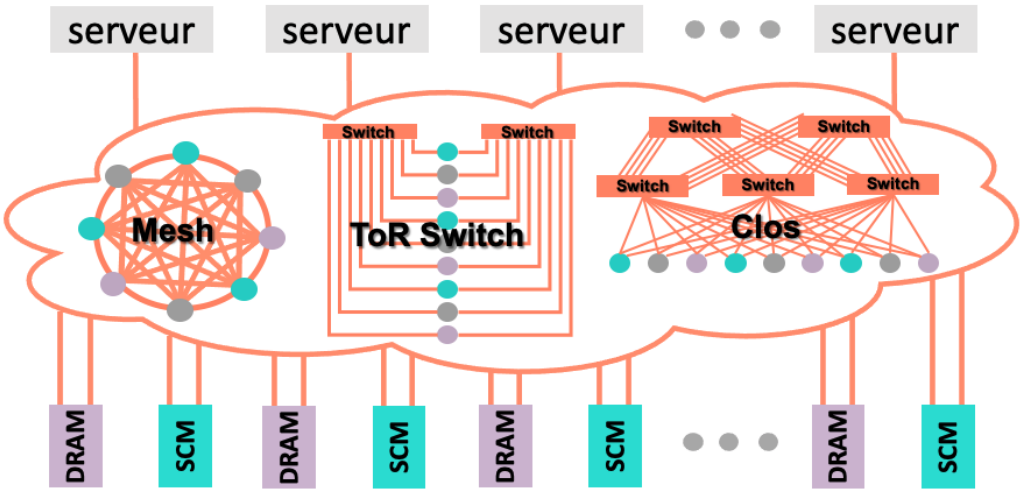


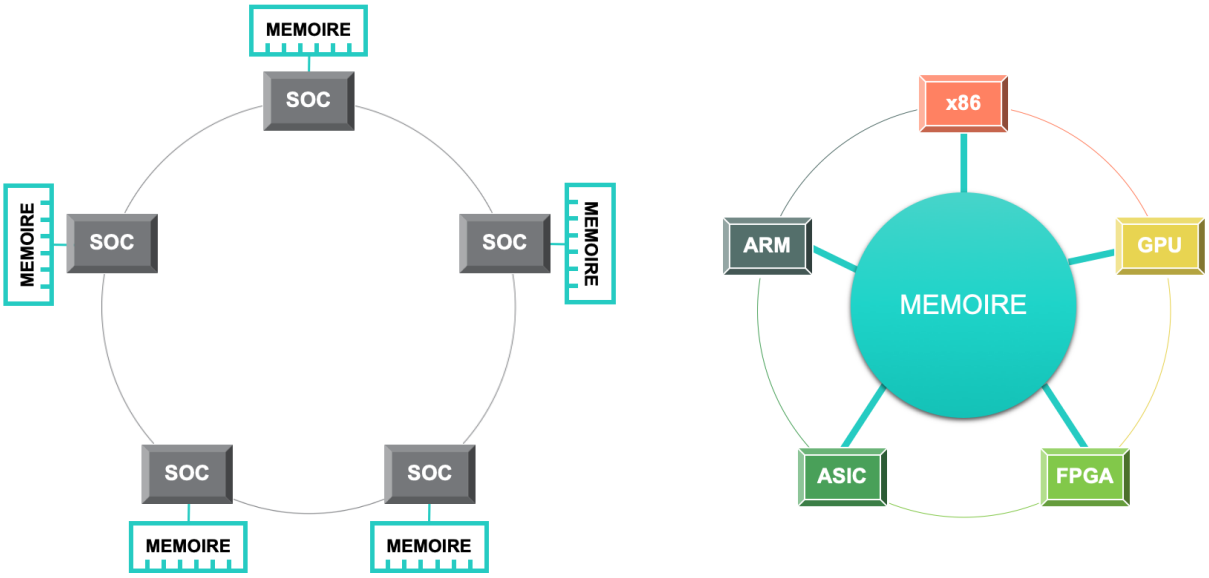
FIGURE 2.52 – Le protocole Gen-Z permet d'implémenter nativement différentes topologies.

et assurée physiquement par chaque matériel grâce à l'utilisation de clefs d'accès. Les accès interdits sont rapportés au gestionnaire pour éventuellement déceler des attaques. De nombreux mécanismes de sécurité sont développés contre les attaques de type *man-in-the-middle*, comme l'utilisation de tags *anti-replay* [RS11].

2.3.5.4 Nouvelles plateformes

Associé aux technologies présentées précédemment comme les mémoires non volatiles (SCM) et/ou les technologies d'interconnexion rapides à faible consommation (photoniques), Gen-Z va permettre de repenser l'architecture des plateformes. Le calcul piloté par la mémoire (Memory Driven Computing (architecture MDC)) désigne une nouvelle façon d'organiser les architectures en plaçant la mémoire au centre (voir figure 2.53). L'architecture MDC donne à chaque processeur l'accès à un grand espace de mémoire partagée (*memory pool*) (figure 2.53b). C'est la différence majeure avec les architectures actuelles où chaque processeur possède son propre espace mémoire local (figure 2.53a). L'avantage de cette architecture est de pouvoir combiner différents éléments de calcul (processeur, accélérateurs) autour de ces espaces mémoire. En fonction des besoins des applications, différentes plateformes peuvent être développées pour répondre précisément au besoin.

Les trois principales caractéristiques dont vont bénéficier les applications sont : le *partage* d'un *grand espace* de mémoire *persistant*. Tous les serveurs partageant le même espace mémoire, il n'est plus nécessaire de réaliser des communications explicites pour échanger des données. Pour le programmeur, cela implique de ne plus avoir à se soucier de partitionner et répartir les jeux de données entre chaque serveur. L'utilisation de **grands espaces mémoire** permet de repenser certains algorithmes. Par exemple en précalculant des graphes pour optimiser leur parcours, ou explorer simultanément plusieurs alternatives. En fonction des motifs d'accès mé-



(a) Plateformes actuelles (b) Plateforme MDC

FIGURE 2.53 – Les plateformes MDC inversent l'architecture en plaçant la mémoire au centre.

moires réalisés, plusieurs versions d'un même jeu de données peuvent être utilisées permettant de profiter des techniques de localités. La totalité des jeux de données pouvant être stockée en **mémoires persistantes**, il n'est plus nécessaire de réaliser des sauvegardes régulières des données sur des périphériques de stockage beaucoup plus lents.

Un même module mémoire pourra être fragmenté et utilisé par différentes ressources. Grâce à Gen-Z, il est possible d'assigner dynamiquement une zone mémoire à une ressource même si celle-ci ne se trouve pas sur le même serveur. Associé à des technologies comme la photonique, l'accès à ces mémoires pourra être très rapide, permettant de créer des espaces mémoire distants de grande capacité. Une zone mémoire peut alors être partagée entre différentes ressources travaillant sur un même jeu de données, ou pour réaliser des communications en mémoire (synchronisation). Lorsqu'un accélérateur doit communiquer des résultats à un second accélérateur, il lui suffit de lui autoriser l'accès à cette zone mémoire et lui transmettre un pointeur associé.

HPE développe depuis plusieurs années une architecture MDC connue sous le nom de code de *The Machine*. Ce projet a donné lieu à la première gamme de serveurs MDC : HPE Superdome Flex. Ces plateformes peuvent embarquer 32 processeurs partageant 48 TB de mémoire. Les premiers résultats ont montré l'énorme saut de performance pouvant être atteint pour certaines applications. Une application de calcul de graphe [Che+16] a ainsi pu être accélérée d'un facteur 128 en utilisant une plateforme de 12 TB de mémoire. HPE utilise des simulateurs permettant de prédire les gains de performances qu'un algorithme peut potentiellement atteindre. Les résultats de ces simulations ont montré que certaines applications utilisées dans le domaine de la finance (simulation Monte-Carlo) pourraient être accélérées d'un facteur 10 000.

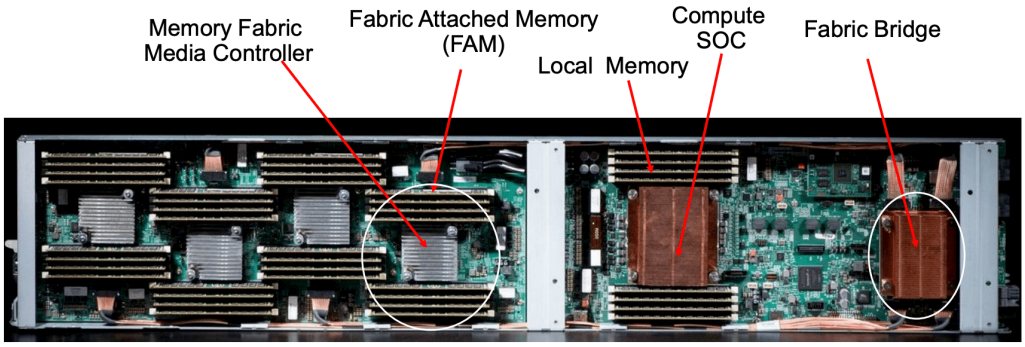


FIGURE 2.54 – Les serveurs HPE Superdome possèdent un espace mémoire *attaché* partagé entre les différents processeurs.

2.4 Caractérisation et analyse de performance des architectures

Dans la section précédente, nous avons discuté des différentes opportunités disponibles pour permettre la construction de plateformes exascale. De nouvelles technologies comme les mémoires SCM ou la photonique vont permettre de développer de nouvelles architectures. Grâce au protocole Gen-Z, différentes architectures pourront facilement être interconnectées dans une même plateforme. L'efficacité énergétique étant primordiale, il est fondamental de trouver les architectures les plus adaptées pour chaque application et d'optimiser leur code pour en tirer la performance maximale. Nous proposons ainsi d'étudier deux domaines permettant de réaliser ce travail : le domaine de la caractérisation d'architecture et le domaine du suivi de performance.

La section 2.4.1 présente les méthodes et les outils existants permettant de caractériser les matériels. Dans un supercalculateur, de nombreuses ressources doivent être caractérisées : le processeur, le système d'interconnexion, le système de stockage... Dans ce travail de thèse, nous nous sommes principalement intéressés à la caractérisation de la microarchitecture des processeurs et du système mémoire :

— Les applications HPC utilisent ces architectures pour exécuter des opérations sur des nombres à virgule flottante (FLOP). Nous nous intéressons donc au matériel responsable de leur exécution : les unités arithmétiques et logiques (ALU).

— La performance de la majorité des applications étant limitée par le système mémoire, nous nous intéresserons dans un second temps à la caractérisation du système mémoire. Ce dernier étant composé d'une hiérarchie de différentes mémoires, il est important de pouvoir caractériser ses différents niveaux : taille, latence, débit.

La section 2.4.2 présente les méthodes et les outils existants permettant de suivre les performances des applications. La complexité des architectures et la différence d'évolution des performances du système mémoire et des processeurs empêchent les applications d'atteindre les performances maximales délivrées par les processeurs.

— Le temps d'exécution des applications HPC est généralement passé dans une faible por-

tion des lignes de codes. Ces zones, appelées points chauds (hot spots), doivent être identifiées pour être caractérisées et optimisées. Pour cela, le programmeur doit utiliser des outils permettant d'obtenir les profils de performances d'une application et identifier ces zones de codes.

— Il est ensuite nécessaire de pouvoir mesurer la performance effective de ces zones de code. Lorsque la performance maximale n'est pas atteinte, il est alors nécessaire de pouvoir en expliquer les raisons.

2.4.1 Caractérisation des architectures

Dans cette partie, nous présentons les différentes méthodes permettant de caractériser les architectures. Nous utilisons le terme *caractérisation* pour désigner la capacité à identifier les forces et les faiblesses d'une architecture ainsi que d'en mesurer certaines caractéristiques. Afin de choisir la meilleure architecture pour une application il est nécessaire de connaître ces différentes caractéristiques : capacité de calcul, performance mémoire (débit, latence)...

Il existe deux façons de caractériser une architecture :

— La première consiste à utiliser les données communiquées par le constructeur d'une plateforme pour calculer la performance du matériel. Dans la section 4.2 nous montrons comment le débit mémoire théorique et la performance de calcul théorique peuvent être calculés à partir des documentations techniques. Cette approche est intéressante, car elle permet de caractériser une architecture sans y avoir accès. Cependant, certaines données peuvent ne pas être disponibles et nécessitent d'avoir accès aux architectures pour les obtenir. Des outils disponibles sur les distributions Linux (*lscpu*[32], *cpumap*[33], *cpuid*[34]) peuvent alors être utilisés pour les obtenir. Néanmoins, ces outils peuvent ne pas supporter toutes les architectures et certaines données peuvent manquer.

— La seconde méthode nécessite d'avoir accès à l'architecture dans le but d'y exécuter des codes dont l'objectif est de mesurer la performance. Ces codes, appelés benchmarks, sont présentés dans la section suivante. Contrairement à la première méthode, l'utilisation de ces applicatifs permet de mesurer une performance réellement atteignable (performance crête). En effet, il est rare que la performance mesurée soit égale à la performance théorique calculée à partir des données techniques de l'architecture. Le développement de nouvelles architectures, et la complexité de leur microarchitecture peuvent donner lieu à des bogues et conduire à de mauvaises performances. Il est donc important de les caractériser pour identifier une erreur, mais aussi pour comparer la performance d'une application avec la performance réelle de l'architecture.

32. http://man7.org/linux/man-pages/man1/lscpu.1.html
33. https://www.plafrim.fr/fr/outils-sgi/
34. https://www.felixcloutier.com/x86/cpuid

2.4.1.1 Benchmarks

En informatique, un benchmark est un code, ou un ensemble de codes, permettant de mesurer la performance d'une solution et d'en vérifier ses fonctionnalités. Lors de la phase de conception d'une architecture, des benchmarks peuvent être utilisés pour détecter la présence de bogues ou pour valider certaines fonctionnalités. Cela permet d'estimer la performance d'un matériel avant qu'il ne soit produit (grâce à des simulateurs par exemple). Les codes de benchmark sont pour la majorité en version libre de droits. Cela permet leur large utilisation et permet de comparer la performance de différentes plateformes (le classement du Top500 est réalisé grâce au benchmark HPL). En informatique, nous pouvons classer les benchmarks selon quatre catégories [Sta05] :

1. **Les benchmarks** sont des applications complètes exécutant différents types d'instructions : calculs, transferts mémoires, réseaux... Le code des benchmarks peut être écrit pour approcher le comportement d'une application existante. Il peut aussi arriver que des applications réelles soient finalement utilisées comme programme de benchmark tel que BSMBench [Ben+14b]. Ce programme est à l'origine utilisé pour réaliser des motifs de calculs similaires à ceux réalisés en théorie de jauge (physique des particules).

2. **Les benchmarks noyaux** (*kernel-based benchmark*) sont des codes simples permettant de caractériser une partie spécifique du matériel. Ces codes artificiels peuvent être de simples extraits de benchmark. Le benchmark noyaux STREAM [McC95] consiste en l'exécution de quatre fonctions différentes qui à l'origine étaient utilisées pour étudier les différences de performances entre deux architectures pour exécuter des applications pour la modélisation du climat. Il est aujourd'hui utilisé pour mesurer le débit mémoire atteignable par un processeur.

3. **Les microbenchmarks** sont des benchmarks noyaux permettant d'isoler une partie précise de l'architecture à évaluer. Par exemple, lmbench [Sta05] est une suite de microbenchmarks portables utilisés pour mesurer des caractéristiques importantes de la mémoire telles que la bande passante, la latence mémoire et les performances des différents niveaux de cache. Les informations récupérées permettent de mettre en oeuvre des optimisations en s'adaptant parfaitement aux caractéristiques d'une architecture.

4. **Les générateurs de benchmarks** permettent de produire une application à partir d'un premier code. La génération a l'avantage de faciliter le test de plusieurs configurations différentes en faisant varier les paramètres d'entrée, la taille des jeux de données, ou encore l'algorithme utilisé pour résoudre une tâche. Ainsi, le logiciel de GeneNetWeaver [SMF11] peut être utilisé pour générer dynamiquement des modèles génétiques pouvant ensuite être utilisés comme benchmark. Des bibliothèques de calcul telles que ATLAS [WD98] et FFTW [FJ98] génèrent des microbenchmarks pour caractériser l'architecture. ATLAS génère ainsi des dizaines de versions différentes d'un code de multiplication de matrices pour estimer la taille de bloc la plus efficace pour exploiter les caches.

2.4.1.2 Benchmarks existants

Comme indiqué dans l'introduction de cette section, nous nous sommes intéressés à la caractérisation du système mémoire ainsi qu'à celle des unités arithmétiques et logiques des processeurs. Ces unités sont responsables de l'exécution des instructions de calcul sur nombre à virgule flottante (FLOP). Elles sont donc un matériel essentiel des architectures pour l'exécution d'application de calcul intensif. Le système mémoire étant le goulot d'étranglement de la performance d'une majorité d'applications, de nombreux travaux ont été réalisés pour sa caractérisation et son optimisation. Dans cette partie nous énumérons et discutons les principaux benchmarks permettant de caractériser ces parties de l'architecture.

HPL [DLP03] Un des micros benchmarks les plus utilisés est sans doute celui du HPL, utilisé pour construire le classement du Top500 [D+97]. Les premières versions du benchmark, alors appelé LINPACK, remontent aux années 1979 et permettaient d'estimer le temps de résolution d'un problème d'algèbre. Son utilisation comme benchmark est plus un accident qu'une réelle volonté. L'annexe B du manuel de l'utilisateur proposait aux utilisateurs d'estimer et de noter les temps de résolution en fonction de la machine utilisée. Pour pouvoir être utilisé sur les supercalculateurs, le benchmark a été parallélisé, changeant ainsi de nom en Highly Parallel Computing Benchmark, HPLinpack ou encore HPL. Il est utilisé pour mesurer le nombre maximum d'opérations à virgule flottante par seconde (FLOPS) qu'un supercalculateur est capable de fournir pour la résolution d'un système linéaire d'équations utilisant la décomposition LU. Ainsi, avant d'être un benchmark de processeur, c'est un benchmark de librairie DGEMM (Intel MKL, netlib, GotoBLAS). La force de ce benchmark est de n'avoir qu'un résultat (soit la sommation des puissances de calcul de tous les coeurs utilisés). Il est donc très facile de comparer deux plateformes de calculs. Bien que mondialement utilisé, le HPL a un principal défaut qui est d'estimer la performance d'une plateforme en ne mesurant que sa capacité de calcul. Cependant, comme nous l'avons montré précédemment, la performance de la majorité des applications est limitée par la performance de la bande passante. La mesure du HPL n'est donc pas la plus représentative de la puissance d'un supercalculateur atteignable par des applications réelles.

HPCG [DH13] Admettant la faiblesse du benchmark HPL, son concepteur, Jack Dongarra, se mit à la recherche d'un code ou d'un ensemble de benchmarks permettant de mieux caractériser ces plateformes. Avec Michael Heroux et Piotr Luszczek, ils présentèrent alors en 2015 le benchmark HPCG (high performance conjugate gradient). HPCG permet de couvrir de nombreux motifs de communication (globale et voisinage) et de calculs (mis-à jour de vecteur, multiplication de matrices creuses). Le premier prérequis était alors de pouvoir produire, grâce à ce nouveau benchmark, un classement des supercalculateurs représentatif de leur performance pour exécuter des applications réelles. Le deuxième prérequis fait suite à la crainte de voir la conception des processeurs être influencée pour obtenir de meilleures performances pour le benchmark HPL [DH13]. Ainsi, HPCG utilise une mesure qui pousserait les concepteurs de

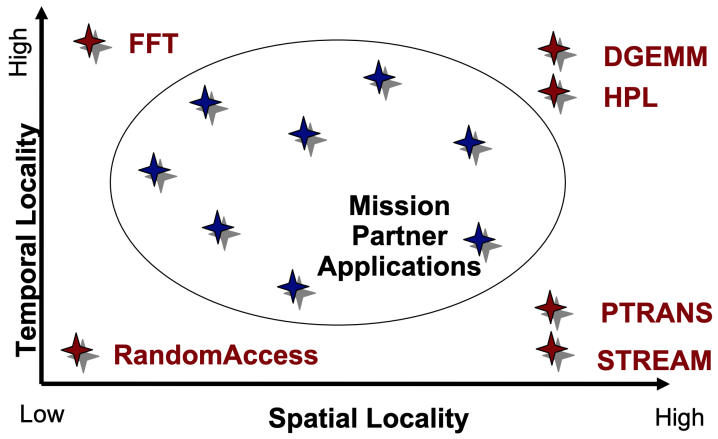


FIGURE 2.55 – La suite de benchmark HPCC utilise des codes utilisant des localités différentes permettant de mieux caractériser les architectures

processeurs à améliorer leurs matériels pouvant mieux profiter aux applications réelles. Là où HPL ne fournit qu'un seul résultat par exécution, HPCG en présente 128. Le classement du Top500 est aujourd'hui publié avec les valeurs obtenues par HPL et par HPCG [35]. Malgré la meilleure adéquation de HPCG à caractériser les plateformes, le benchmark HPL est, et sera probablement toujours utilisé, principalement pour des raisons d'historiques. En effet, il permet de suivre l'évolution des architectures depuis plus de 25 ans.

HPCC [Lus+06] La suite de benchmark HPC Challenge vient compléter le benchmark HPL avec 6 codes dont certains réalisent des accès aux données permettant aussi de caractériser le système mémoire (voir figure 2.55). La suite de benchmarks est ainsi composée de HPL, Stream, DGEMM, RandomAccess, b_eff, PTRANS, FFT. Avec l'ajout de ces six autres codes, la suite HPCC est plus représentative des applications réelles. Ces codes existaient avant la création de la suite HPCC. Le travail réalisé a permis de les regrouper dans une même suite, et de les améliorer avec des systèmes de vérifications et de rapport. Une fois compilé, un seul programme exécutable est généré permettant de faciliter son usage avec un gestionnaire de *job*. Un seul exécutable étant généré, l'environnement d'exécution est le même pour tous les benchmarks de la suite (pas d'optimisation de pages larges pour un seul benchmark de la suite).

SPEC CPU2017 [BLK18] Cette nouvelle génération de benchmark produite par SPEC (Standard Performance Evaluation Corporation) qui fait suite à la version précédente SPEC CPU2006. Ces deux initiatives ont permis de regrouper des applications réelles de tailles différentes, mais dont les performances sont limitées par la puissance de calcul de l'architecture. Les applications sélectionnées sont facilement portables. La première version contenait deux suites de benchmarks (CINT2006 [Dil12] et CFP2006 [Sha+09]) permettant de mesurer et comparer les performances de calculs en utilisant des opérations entières ou à nombres flottants. L'objectif

35. Classement HPCG 2019 : https://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=302

principal de ces codes était alors de caractériser les performances du processeur, de la hiérarchie mémoire et du compilateur. La version 2017 possède 43 benchmarks qui sont portés sur plusieurs architectures dont AMD64, Intel IA32, Power ISA ou SPARC. Les benchmarks sont organisés en quatre suites permettant de mesurer le débit et la vitesse d'exécution d'opérations utilisant des nombres entiers ou flottants. Les différents benchmarks ont un domaine d'application spécifique (compression vidéo, rendu 3D...) et peuvent être utilisés pour la conception de processeurs optimisés pour ces charges de travail [Pan+18]. Malheureusement le prix de ces benchmarks avoisine les 1000$. Cependant les nombreux résultats, libres de droits, sont publiés leur site internet.

STREAM [McC95] Le benchmark STREAM est sûrement un des benchmarks les plus connus et les plus utilisés au monde. Il a été développé et est maintenu par John McCalpin surnommé "Dr. Bandwidth". Le code STREAM permet de mesurer la bande passante mémoire atteignable grâce à l'implémentation de quatre noyaux : COPY ($c = a$), SCALE ($b = \alpha \times c$), ADD ($c = a + b$) et TRIAD ($a = b + \alpha \times c$). Les résultats sont donnés en GB/s et contiennent à la fois les opérations de lecture et d'écriture. Pour ces quatre opérations, STREAM fonctionne en générant un tableau de nombres aléatoires d'une taille spécifiée (qui est ensuite stocké en RAM) et effectue quatre types d'opérations : *copy, scale, add, triad*. Le benchmark utilise *OpenMP* pour utiliser la totalité des coeurs disponibles. Ces différents tests étaient à l'origine destinés à caractériser la performance des architectures vectorielles. La performance mémoire pouvait alors varier d'une opération à l'autre. Aujourd'hui, la performance calculatoire des architectures n'est plus la contrainte principale et les quatre microbenchmarks obtiennent des performances équivalentes. Il est généralement accepté que la mesure donnée pour l'opération de *triad* correspond à la bande passante maximale atteignable par l'architecture. On remarque que le noyau de calcul du *triad* est relativement simple et ne consiste qu'en la lecture de deux éléments et l'écriture du résultat. Les applications réelles utilisant des motifs d'accès bien plus complexes, cette mesure n'est pas représentative de la performance réellement atteignable par celles-ci [36].

Lmbench [Sta05] Le benchmark `Lmbench` [Sta05] a été développé par deux ingénieurs des HP Labs d'Israël en 2004. Ce code est en fait une suite de microbenchmarks permettant de mesurer la performance de plusieurs aspects d'une architecture : lecture d'un jeu de données, ouverture de fichiers, création de pipe, fréquence mémoire, taille d'une ligne de cache, taille de la TLB, bande passante mémoire (STREAM). L'ensemble des codes peut être exécuté pour caractériser la mémoire d'un système partagé ou distribué [SI02].

`Lmbench` facilite l'ajout de nouveaux microbenchmarks et mesure leurs performances en donnant la latence par instruction et le débit mémoire. Le framework s'occupe de leur exécution pour atteindre des mesures de performances ayant une performance d'au moins 1%. Écrit en

36. https://www.intel.ru/content/dam/doc/white-paper/resources-xeon-7500-measuring-memory-bandwidth-p
pdf

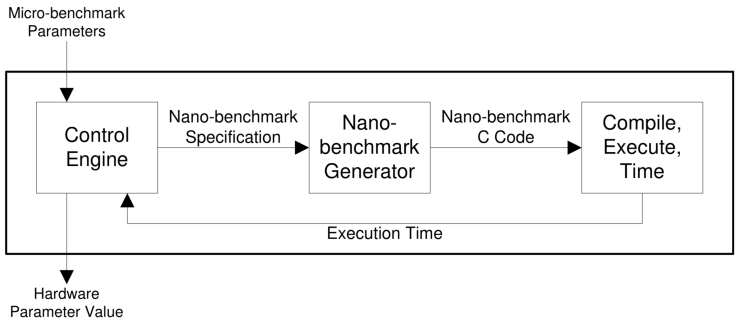


FIGURE 2.56 – Structure d'un microbenchmark réalisé grâce à X-Ray [YPS04].

ANSI-C et respectant la norme POSIX, le benchmark a été développé pour maximiser sa portabilité. Cependant, sa compilation sur des architectures récentes peut être plus difficile [YPS04]. En raison de son incapacité à mesurer les performances du cache distant et les transactions de cohérence du cache, le benchmark *x86-membench* benchmark [Mol+17] a été développé pour supporter la mesure de la bande passante et de la latence du cache local ou distant, mais aussi de la mémoire. Le benchmark n'utilise aucune méthode de parallélisme empêchant une caractérisation poussée des architectures modernes.

X-Ray [YPS04] Pour aider les programmeurs à écrire de nouveaux benchmarks permettant de mesurer certains paramètres matériels l'outil X-Ray a été mis au point. X-Ray est un framework permettant l'automatisation du développement de benchmarks. Pour cela il génère plusieurs benchmarks grâce à un framework *Nano-benchmark Generator* (figure 2.56). Cette génération est dynamique, car les benchmarks à générer varient en fonction de ceux déjà exécutés. Si un benchmark a besoin de connaître la latence mémoire d'un niveau de cache, X-Ray exécutera alors le benchmark permettant de le calculer avant. X-Ray peut par exemple calculer la fréquence du processeur en ajoutant quatre nombres entiers avec des dépendances pour éviter les optimisations des processeurs superscalaires. Des benchmarks sont aussi disponibles pour caractériser la hiérarchie mémoire (associativité, taille des blocs, capacité, latence). Les résultats présentés sont annoncés plus précis que ceux donnés par les autres outils de l'état de l'art. Différents tests ont été réalisés sur des ordinateurs personnels, des serveurs ou encore des systèmes embarqués. Malheureusement le code n'est plus disponible pour être testé.

Le travail [YPS05] utilise X-Ray pour implémenter des micros benchmarks pour mesurer la capacité, l'associativité, la taille des blocs ainsi que la latence de chaque niveau de la hiérarchie de caches ainsi que du TLB. Contrairement aux benchmarks existants, l'outil mesure un niveau à la fois, lui permettant d'être plus précis que les approches traditionnelles (X-Ray, Calibrator, lmbench et MOB). Utiliser X-Ray pour implémenter leur benchmark leur permet de mesurer la fréquence du processeur ainsi que la latence et le débit des instructions. Malheureusement, le code de X-Ray n'a pas été maintenu et n'est plus disponible au téléchargement.

P-Ray [DS08]. Pour remédier à l'incapacité de `Lmbench` à caractériser les plateformes multicoeurs, le benchmark P-ray a été développé. Pour cela il étend les micros benchmarks existant pour trouver le niveau des caches partagé, la topologie d'interconnexion, la bande passante effective ou la taille des blocs pour la gestion de cohérence des caches. Pour éviter les optimisations du compilateur (*pointer chaising*), le benchmark utilise un système de listes chaînées lors de l'initialisation. Les résultats obtenus sont eux très précis et s'approchent souvent des maximums théoriques attendus.

Dans le but de permettre à certaines librairies (ATLAS, SPIRAL ou FFTW) de s'auto-optimiser, P-Ray permet de mesurer certains paramètres matériels. De la même façon que X-Ray et LMbench, l'apport de P-Ray permet de trouver ces spécifications pour des processeurs multicoeurs. P-Ray permet de décrire la répartition des caches entre les coeurs, la topologie d'interconnexion des processeurs ainsi que les mécanismes de cohérence de cache. Leurs expérimentations montrent des résultats très précis comme la mesure de la latence de communication entre deux coeurs à travers le cache L3 ou entre deux processeurs. Un effort particulier a été apporté pour éviter des optimisations du compilateur et du préchargement mémoire (*memory prefetcher*) pouvant altérer les performances mesurées. Le benchmark P-Ray a hérité d'un problème de X-Ray rendant impossible la portabilité du code entre différents systèmes d'exploitation. L'allocation mémoire suppose que toutes les adresses physiques soient contiguës et cette caractéristique dépend du système d'exploitation (les pages larges pouvant ne pas être disponibles). Le code de P-Ray n'est cependant pas libre de droits.

Servet [Gon+10]. La suite Servet permet de mesurer certaines caractéristiques des matériels, telles que la hiérarchie de cache (taille, partage entre les coeurs) ou la bande passante mémoire. Ce travail complète X-Ray et P-Ray par des mesures de paramètres d'interconnexion pour la communication d'une mémoire distribuée. Une suite de benchmarks permet aussi d'évaluer où se formeront les goulots d'étranglement lorsque plusieurs coeurs accèdent à la mémoire centrale. Enfin, Servet mesure la distance entre les coeurs en mesurant la latence de communication permettant à une application de placer plus efficacement les processus sur les différents coeurs.

Saavedra [SS95] Ce benchmark utilise une stride (un saut en mémoire) de taille fixe pour accéder aux éléments d'un tableau. Le temps nécessaire à ces accès permet de déduire la taille des niveaux de la hiérarchie. Les expérimentations utilisent des couples de $\{tailledetableau, tailledestride\}$. La taille du tableau augmente jusqu'à atteindre la taille du cache mesuré. La taille des strides est limitée à des tailles de puissance de 2 et ne dépassera jamais la taille du cache. Le tableau est ainsi parcouru pendant au moins une seconde. Comme le souligne [YPS05] le problème d'une telle approche est de vouloir mesurer tous les niveaux de la hiérarchie simultanément. Les mesures peuvent alors être influencées par différents paramètres de différents niveaux de caches. Ces mesures doivent être interprétées par l'utilisateur, le programme ne créant pas automatiquement la hiérarchie. La lecture et l'écriture sont réalisées sur la même donnée pouvant introduire des conflits dans le tampon d'écriture [YPS05]. Le benchmark suppose que les

données sont continues en mémoire, mais n'utilise pas de pages larges.

2.4.2 Analyse de performance d'une application

Le suivi de performances (*performance monitoring*) a pour but de récolter des informations concernant une application ou concernant le système sur lequel elle est exécutée, pour la déboguer ou l'optimiser. L'analyse de performance peut être réalisée à différents niveaux [Ima+11] :

— Le premier niveau concerne l'utilisation d'un simulateur permettant d'étudier précisément (cycle par cycle) le comportement d'une architecture. Les données collectées peuvent être riches et permettent d'apporter des informations qu'il ne serait pas possible d'avoir avec l'exécution réelle de l'architecture. L'avantage de cette méthode est de pouvoir réaliser des premiers tests, sans avoir accès à l'architecture. Ces simulations peuvent modéliser la totalité de la microarchitecture et suivre l'exécution d'une application. Cependant, cette méthode nécessite le développement de simulateurs complexes que seul le constructeur peut développer. Les simulations précises (au cycle près) sont lentes (quelques centaines de cycles simulés par seconde) et génèrent de grandes quantités de données difficiles à analyser [Pal15]. Pour y remédier, des simulateurs plus simples ayant recourt à des traces générées sur une architecture réelle peuvent être utilisés [CK95]. Les traces sont collectées lors d'une première exécution sur une architecture et sont rejouées dans un simulateur pour être étudiées.

— Le second niveau correspond à l'analyse de la performance d'un serveur (coeur, processeur, système mémoire, réseaux). L'analyse de performance peut consister à vérifier la bonne répartition du travail entre les processeurs et la bonne utilisation des coeurs. Certains problèmes de performance peuvent venir de la mauvaise utilisation des mémoires caches ou de celle du bus mémoire. Au niveau le plus fin, l'analyse de la performance des coeurs peut être difficile, car elle nécessite l'utilisation d'outils précis. La complexité de la microarchitecture rend d'autant plus difficile cette analyse, de solides connaissances sont nécessaires pour pouvoir les analyser. Suite à cette analyse, diverses optimisations peuvent être nécessaires : préchargement de la mémoire, utilisation d'instructions vectorielles, amélioration de la gestion de la localité des données (section A.3.2.3)... Les informations utilisées pour l'analyse peuvent être obtenues en instrumentant le code (manuellement ou grâce au compilateur) ou en récupérant certaines informations de l'architecture (compteurs matériels).

— Le dernier niveau concerne l'analyse de la performance du supercalculateur complet. Il peut s'agir de vérifier la bonne utilisation de la programmation distribuée et la répartition du travail entre les noeuds. Les supercalculateurs exascale devront être hétérogènes. Il est donc capital que l'ordonnanceur de tâche (*job scheduler*) place efficacement les différents processus.

Pour parvenir à extraire la performance des supercalculateurs, l'analyse et l'optimisation des applications doivent être faites aux trois niveaux. Ce travail de thèse s'intéresse à l'analyse

de performance du second niveau. Afin de permettre l'analyse, le portage et l'optimisation des codes, deux étapes principales sont étudiées :

1. **Identifier les hot spots.** En 1971, Donald Knuth affirmait que la majorité de l'exécution d'une application se déroulait dans une fraction des lignes de codes [Knu71]. Cette affirmation est aussi connue sous le nom de principe de Pareto (règle du 80/20). Ce principe empirique postule que 80% des effets sont le produit de 20% des causes. Dans le domaine de l'analyse de performance, cela signifie que 20% des lignes de code sont responsable de 80% du temps de l'exécution de l'application. Ces zones de codes sont appelées points chauds ou hot spots. Cela est d'autant plus vrai pour les applications de calcul haute performance où la proportion de code responsable d'une grande partie de la performance peut être encore plus faible. Il est facile de comprendre que c'est l'optimisation de ces zones de codes qui aura le plus d'impact sur l'amélioration de la performance de l'application. Il est donc nécessaire de pouvoir identifier et caractériser ces zones.

2. **Identifier les goulots d'étranglement.** Une fois les zones de codes prenant une part majeure dans le temps d'exécution de l'application, il est nécessaire de quantifier leur performance pour trois raisons :
   — La première est de mesurer l'écart entre la performance réelle de l'application et la performance théorique atteignable par l'architecture pour quantifier les opportunités d'optimisations.
   — La deuxième est de pouvoir identifier les raisons et les parties de la microarchitecture qui sont responsables de cette performance.
   — La troisième est de permettre de caractériser leurs besoins : latence et débit mémoire, calcul, stockage...

   Pour pouvoir comparer la différence entre la performance théorique et celle mesurée lors de l'exécution de l'application, il est courant d'utiliser des modèles de performances. Ces modèles prennent en compte les caractéristiques techniques d'une microarchitecture (FLOPS, débit mémoire...) ainsi que celles de l'application. L'utilisation de tels modèles permet ensuite de savoir quand le travail d'optimisation est terminé (performance maximale atteinte) ou de prédire la performance de la même application sur un système différent.

Afin de répondre aux besoins exprimés ci-dessus, la suite de cette section présente les différentes méthodes et les outils existants qui permettent de suivre la performance d'une application. Nous présentons ensuite un modèle simple permettant de quantifier les performances mesurées d'une application.

2.4.2.1 Analyse statique et dynamique

Afin d'obtenir les informations nécessaires pour l'analyse de la performance d'une application, deux approches peuvent être employées : l'analyse statique et l'analyse dynamique.

L'analyse statique L'analyse statique consiste à obtenir des informations sur une application sans avoir à l'exécuter. Cette analyse peut permettre de détecter des bogues [Lat16] ou de modéliser la performance de l'application grâce à l'utilisation de modèles analytiques. L'analyse peut être réalisée à partir du code source, ou des instructions assembleur générées par le compilateur [DB05 ; Won+15]. L'utilisation de l'assembleur permet de valider le bon fonctionnement du compilateur [Cha+14], mais rend plus difficile la corrélation avec le code source [De 10].

Des outils existent et permettent de réaliser des analyses statiques d'applications. Intel propose différents outils tel que Intel Architecture Code Analyzer (IACA) [Hir12] qui permet de réaliser une analyse statique d'un code grâce à l'ajout de marqueurs dans le code source. Il permet, pour un noyau identifié, de détecter la présence de dépendance entre plusieurs itérations de boucle et de donner une estimation des performances (débit d'instructions, saturation des ports de l'ALU...). Cependant, il est nécessaire d'avoir identifié les zones de *hot spots* pour les annoter, et il ne fonctionne que pour des architectures Intel. Le projet a été abandonné en avril 2019 [37].

L'outil Modular Assembly Quality Analyzer and Optimizer (MAQAO) [DB05] permet de désassembler un binaire pour en extraire le code assembleur et de lister les fonctions et les boucles importantes de l'application. Il peut ensuite faire un retour à l'utilisateur sur la qualité de son code et lui donner des conseils pour l'améliorer. Il permet d'analyser différents points, dont les 4 présentés dans [Pop16] :

- La pression appliquée par les instructions sur les différents ports du processeur pour en déterminer le bottleneck.
- La performance crête d'un code (qui suppose que toutes les données sont disponibles dans le premier niveau de cache).
- L'intensité arithmétique en mesurant le rapport du nombre d'instructions réalisant des opérations sur des nombres flottants et le nombre nombre total d'instructions à exécuter.
- Le ratio de vectorisation en mesurant le rapport du nombre d'instruction utilisant la vectorisation sur le nombre d'instructions pouvant potentiellement l'être.

L'analyse statique ne nécessite pas l'exécution de l'application pour être réalisée. Ceci est très avantageux pour analyser les applications HPC dont l'exécution peut durer plusieurs heures. Malheureusement, l'absence d'exécution ne permet pas de détecter les nombreux problèmes de performances dus à la complexité de l'architecture. Il est donc très difficile d'anticiper la performance réelle des applications.

L'analyse dynamique Contrairement à l'analyse statique, l'analyse dynamique nécessite l'exécution de l'application pour évaluer sa performance. Elle permet d'apporter de nombreuses informations impossibles à avoir grâce à l'analyse statique. Les informations peuvent être obtenues grâce à l'implémentation matérielle de registres permettant de suivre l'activité de la microarchitecture. Ces registres peuvent être simples (temps écoulé) ou complexes (nombre d'instructions vectorielles exécutées). Il y a deux façons d'utiliser ces compteurs : le comptage et l'échantillonnage.

37. Intel IACA - https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

— La première méthode consiste à **compter** le nombre d'évènements arrivant entre deux intervalles de temps. Pour cela, le compteur est initialisé à 0 et lu au bout d'une certaine période de temps. Les valeurs ainsi récupérées permettent de mesurer le nombre d'occurrences de ces évènements. Cette méthode est efficace, mais elle ne permet pas de connaître la partie du code responsable d'un évènement. Les ratios d'évènements, tels que les instructions exécutées par cycle, les taux de *miss* de mémoire cache et les taux d'erreurs de prévision des branches, peuvent être calculés en divisant le nombre par le temps écoulé.

— La deuxième méthode consiste à **échantillonner** ces informations. Ce mode consiste à déclencher une interruption tous les n évènements et sauvegarder certaines informations telles que le pointeur d'instruction. Pour cela, le registre de comptage est initialisé à la valeur $\text{MAX} - n$ ou MAX correspond à la valeur maximale pouvant être stockée dans le registre. Lorsque n évènements sont comptés, le registre déclenche un débordement (*overflow*) et génère une exception traitée par le système d'exploitation. Grâce à un échantillonnage assez fin (nombre n petit) et des méthodes de statistiques, il est possible d'approcher le nombre d'évènements généré par chaque instruction. La principale difficulté de cette technique est d'assurer suffisamment de précision lors de l'attribution d'un évènement à une instruction. En effet, entre le moment où l'interruption est générée et son traitement, plusieurs instructions peuvent avoir été exécutées. Des technologies telles que Intel PEBS [38] (Processor Event-Based Sampling) ou AMD IBS (Instruction Based Sampling) [Dro07] agrémentent le processeur d'un tampon lui permettant de stocker les informations nécessaires. L'autre avantage de cette technologie est de réduire l'impact sur les performances dû au traitement individuel de chaque échantillon par le système d'exploitation. La PMU possède un tampon pouvant stocker plusieurs échantillons et n'interrompt l'exécution que lorsque ce tampon est plein. Le principal désavantage de cette technologie est le nombre restreint d'évènements compatibles. De plus, elle n'est pas compatible avec toutes les architectures réduisant la portabilité des outils l'utilisant.

2.4.2.2 Les compteurs matériels

Afin de fournir des informations pour permettre l'analyse de performance dynamique, la microarchitecture possède des registres spéciaux qui sont incrémentés lors du déclenchement d'un évènement. Ces registres sont appelés compteurs matériels et leur origine est expliquée dans l'Annexe B. Les évènements mesurés peuvent être matériels (cycle d'horloge, exécution d'une instruction, manque dans un cache...) ou logiciels (faute de page, changement de contexte...). Suivre l'évolution de ces compteurs peut alors être utilisé pour des raisons multiples [MVJ11] : optimisation guidée par profil [Cav+06], optimisation dynamique [Dai+05], adaptation de la gestion de l'énergie [ICM05], ordonnancement des *threads* pour les ressources partagées [Mos06] et autres [Shy+05 ; Shy+08].

38. Documentation Intel - Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, Chapter 18. https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf

L'avantage des compteurs matériels est d'être implémentés directement par le processeur sans quoi il serait impossible de suivre précisément l'activité de la microarchitecture (*miss* dans les caches, remplacement dans la TLB, nombre de cycles exécutés). Il faudrait pour cela utiliser des simulateurs dont les résultats sont souvent différents des performances réelles des processeurs réels. La tâche du programmeur est de configurer les compteurs et consulter leur valeur pour réaliser son étude. Cependant l'utilisation des compteurs matériels a plusieurs inconvénients :

— **Programmation des compteurs** : Afin de pouvoir utiliser les compteurs, le programmeur doit connaître leur adresse et être capable d'encoder les évènements pour pouvoir les programmer. Ces informations sont disponibles dans les documentations fournies par les constructeurs de processeur et ne sont pas toujours complètes (ou protégées par des accords de non-divulgation). Le programmeur doit ensuite développer un code de bas niveau pour programmer individuellement chaque compteur. Nous montrons dans la section B.2.1 comment ceci peut être réalisé et démontrons la difficulté d'utiliser ces compteurs.

— **Validation des évènements** : Pour pouvoir tirer des conclusions des résultats mesurés par les compteurs, il est nécessaire de réaliser deux validations : la première est de s'assurer que l'évènement utilisé mesure bien le comportement souhaité, la deuxième est de valider le bon comportement du compteur matériel. Les évènements disponibles pour un processeur peuvent varier entre deux versions de processeurs rendant le développement d'outil très difficile. Intel propose la liste des évènements compatibles avec chaque architecture [39].

— **Interprétation des résultats** : L'interprétation des mesures obtenues est une difficulté majeure dans l'utilisation des compteurs matériels. Interpréter un évènement comme le "*nombre de miss par cycle dans le cache L1*" est loin d'être trivial. Il faut être capable de définir un seuil au-delà duquel la valeur obtenue indique une mauvaise performance. Associer des *bonnes* et *mauvaises* valeurs aux évènements est un défi [MVJ11].

Choix d'un sous-ensemble d'évènements. Les avantages et les inconvénients résumés ici étaient pour la majorité déjà actuels en 2002 [Spr02]. Ce domaine ayant peu évolué depuis toutes ces années, nous devons nous satisfaire de l'état actuel de ces technologies sans espérer d'amélioration. Les outils développés dans le cadre de cette thèse ont pour objectif d'être utilisés sur des architectures très différentes. Pour cette raison, nous avons décidé de restreindre le nombre d'évènements nécessaires pour leur fonctionnement et nous privilégions l'utilisation de compteurs accessibles sur la majorité des plateformes. Ces compteurs sont généralement les plus simples et les plus robustes : nombre de cycles et nombre d'instructions exécutées. Grâce à ces deux compteurs, le nombre d'instructions exécutées par cycle peut être calculé (IPC). Ce ratio est une donnée primordiale pour s'assurer de la performance d'un code, mais elle n'est pas suffisante. Un axe central de notre méthodologie de caractérisation des applications nécessite de pouvoir suivre l'activité du bus mémoire. Les compteurs permettant de compter distinctement les accès en lecture et écriture réalisés par chaque contrôleur mémoire

39. Liste des évènements compatibles par architecture Intel - https://download.01.org/perfmon/index/

sont alors nécessaires. L'expérience acquise durant ces travaux de thèse nous a montré que ce sous-ensemble d'évènements était suffisant pour poursuivre l'analyse de la performance d'une grande majorité d'applications.

2.4.2.3 Interface pour utiliser les compteurs matériels

La totalité des supercalculateurs HPC utilise un système d'exploitation basé sur le noyau Linux. Nous nous intéressons dans cette partie aux différents moyens disponibles lors de l'écriture de cette thèse pour utiliser les compteurs qui sont compatibles avec cet environnement. Dans l'Annexe B nous présentons les différentes façons de programmer les compteurs :

— Instructions assembleurs `wrmsr`, `rdmsr` et `rdpmc` (section B.2.1)
— Interface `/sys` (section B.2.2.1)
— Commande `lspci` et `setpci` (section B.2.2.2)
— Mappage de l'espace mémoire PCI (section B.2.2.3)

Ces méthodes nécessitent de solides connaissances de l'architecture étudiée. De plus, le code développé est très dépendant de l'architecture rendant difficile le développement d'outils sophistiqués et leur maintien sur différentes versions d'architectures. Afin d'absoudre l'utilisateur de ces difficultés, des interfaces plus haut niveau ont été développées. La suite de cette section présente ces différentes interfaces (voir figure 2.57).

Perfmon2 Le support par Linux des compteurs matériels a été long, empêchant les développeurs de construire des outils facilement portables. Pour ce faire, différents *patchs* logiciels devaient être appliqués au noyau pour supporter leur utilisation. Nous pouvons citer *perfctr* [Pet05] ou les projets *perfmon* et *perfmon2* [Era06], développés par Stéphane Eranian, ancien employé HP. Le projet *perfmon2* était le principal candidat à l'inclusion dans le noyau Linux avant l'émergence de *Perf Events*. L'objectif du projet *perfmon2* est de concevoir une interface générique de suivi des performances de bas niveau pour accéder à toutes les implémentations d'unités de surveillance des performances matérielles (PMU). L'interface utilise l'appel système *perfmonctl*. Un appel système a été préféré à un driver de périphérique, car il offre une plus grande flexibilité et facilite la prise en charge par les implémentations d'un mode de surveillance par thread qui nécessite d'enregistrer et de restaurer l'état PMU du thread. L'interface exporte tous les registres PMU comme des registres 64 bits, même si de nombreuses implémentations de PMU en ont moins. L'interface ne sait pas ce que fait chaque registre, leur nombre, ni comment ils sont associés les uns aux autres. Toutes les informations spécifiques à un événement sont reléguées au niveau de l'utilisateur où elles peuvent être facilement encapsulées dans une bibliothèque.

Le code de *perfmon2* n'a jamais réussi à être inclus dans le projet Linux et son développement a été arrêté suite à la présentation du projet concurrent *Performance Counters for Linux* (PCL) rebaptisé depuis *Perf Events*.

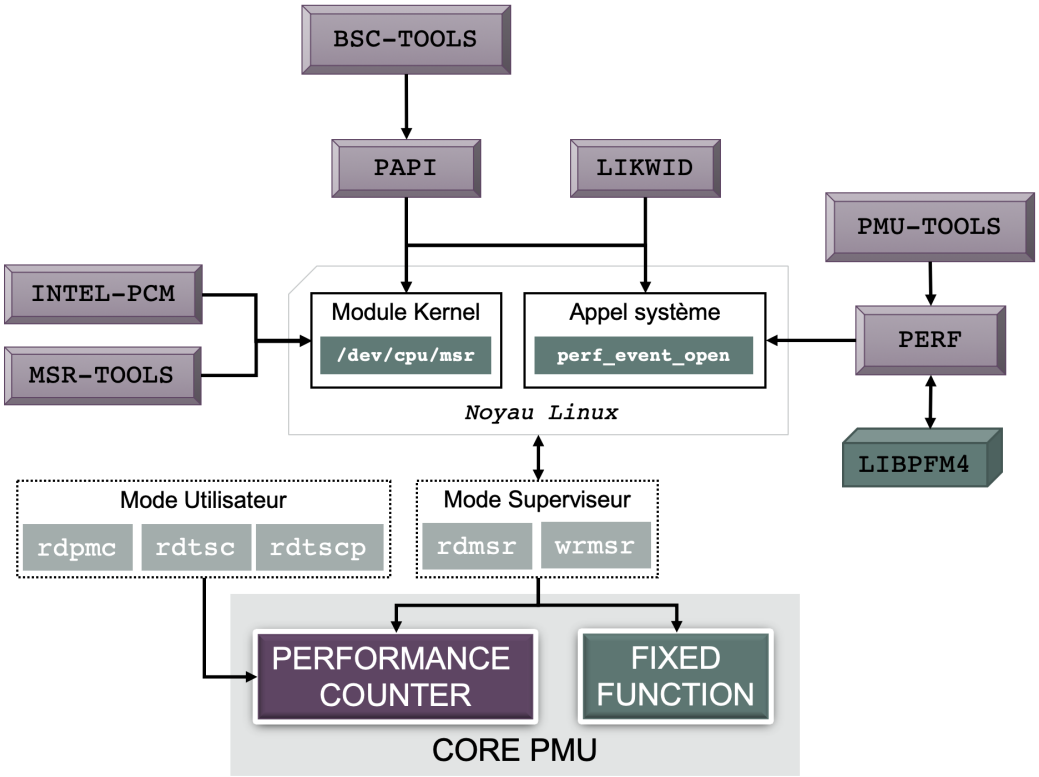


FIGURE 2.57 – Différents moyens d'accéder aux compteurs localisés sur la PMU des coeurs. Linux propose deux interfaces permettant d'utiliser les deux instructions `rdmsr` et `wrmsr` : un module noyau et un appel système.

Perf Events Le projet de Compteur de Performance pour Linux (PCL) a été présenté en décembre 2008 et introduit dans Linux 2.6.31. Cet outil, rebaptisé *Perf Events*, est une réponse au projet principal de standardisation des compteurs pour Linux de l'époque : perfmon2. Alors que ce dernier présentait beaucoup de fonctionnalités intéressantes, il n'a jamais réussi à être inclus dans le projet Linux. Lorsque *Perf Events* fut ajouté au noyau Linux, de nombreuses fonctionnalités manquaient, mais sont apparues depuis. Pour fonctionner, *Perf Events* utilise son interface *perf_event* développée directement dans le code du noyau Linux. Son adoption dans le noyau a mis un frein aux développements d'autres interfaces telles que *perfmon2* ou *perfctr* qui nécessitait de patcher le noyau pour être utilisées. Du projet *perfmon2*, seule la librairie *libpfm4* est encore active. Il s'agit d'une librairie qui permet de convertir les noms symboliques d'évènements PMU en leur encodage pour *Perf Events* ou d'autres interfaces noyau (exemple de la figure 2.58). La librairie en elle-même ne mesure aucun évènement. Elle fournit aux développeurs une interface pour lister, encoder les évènements pour beaucoup de PMU (core et uncore). Un point de désaccord persistant entre *Perf Events* et d'autres approches (telle que *perfmon2*) est qu'il incorpore toutes les informations sur les compteurs et les événements dans le code du noyau lui-même, ajoutant ainsi une quantité significative de code descriptif au noyau. L'intégration de telles données dans le noyau peut causer des difficultés aux outils utilisateurs

pour prendre des décisions sur les événements qui peuvent être comptés simultanément et aux fournisseurs pour fournir des mises à jour sans avoir à patcher le noyau ou attendre de nouvelles versions du noyau. L'interface introduit un seul appel système comme point d'entrée (*sys_perf_open*) qui renvoie un descripteur de fichier qui donne accès aux événements matériels.

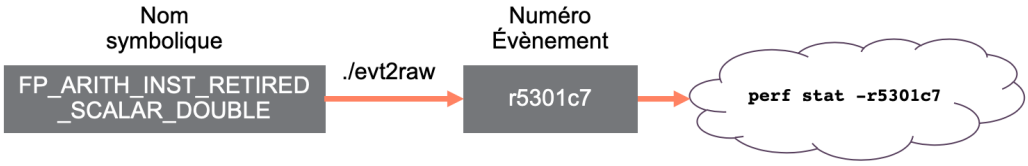


FIGURE 2.58 – *Libpfm4* permet d'obtenir l'encodage d'un évènement à partir de son nom symbolique. Ce code peut ensuite être utilisé avec des outils tels que *perf*.

Pseudo système de fichier `/dev/cpu/CPUID/msr` Linux permet à l'utilisateur un accès bas niveau aux MSR (registres matériels) sans avoir à utiliser de langages assembleurs. Ceci est réalisé à travers un pseudo système de fichier permettant d'interagir avec tous les coeurs (identifié par leur `CPUID`) à travers le chemin `/dev/cpu/#CPUID/msr`. Par défaut, seul l'utilisateur *root* est autorisé à modifier ce fichier, mais ces droits peuvent être modifiés pour étendre l'accès aux autres utilisateurs. L'utilisation de cette interface est très pratique et peut être réalisée en accédant au fichier *msr* grâce aux opérations `pread()` et `pwrite()`. Par exemple, pour lire le MSR dont l'adresse est `0x38`, il suffit d'ouvrir le fichier *msr* correspondant au coeur voulu, et réaliser une lecture avec un décalage de 38 octets. Pour faciliter l'usage de l'interface, Intel propose l'outil *msr-tools* sur son GitHub [40] (voir figure 2.59). La compilation produit deux exécutables `rdmsr` et `wrmsr` qui permettent de réaliser les lectures et écritures des registres MSR comme cela : `./rdmsr -r 0x38D`. Nous proposons dans notre travail trois scripts permettant d'initialiser et configurer les compteurs grâce à ces deux programmes [41].

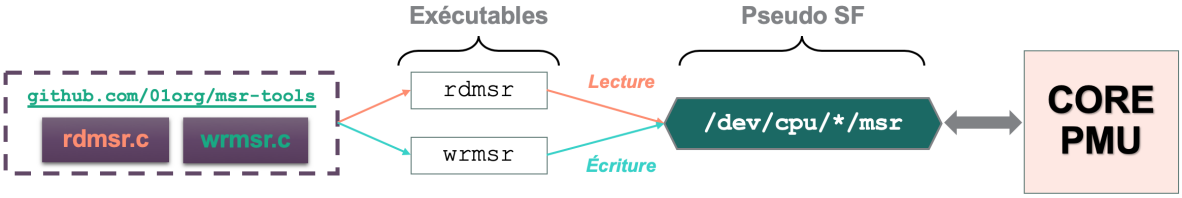


FIGURE 2.59 – Intel propose deux exécutables permettant d'interagir plus facilement avec le pseudo système de fichier.

40. https://github.com/intel/msr-tools
41. https://github.com/PourroyJean/performance_modelisation/tree/master/src/tool_PMU

2.4.2.4 Outils d'analyse de performance

Les interfaces présentées précédemment permettent de faciliter l'accès et la programmation des compteurs matériels. À l'aide de ces interfaces, de nombreux outils de suivi de performance ont pu être développés. Dans cette section, nous présentons les principaux outils utilisés pouvant avoir une utilité dans le travail fixé au début de cette partie.

perf [Wea13]. En plus de l'appel système, *Perf Events* fournit un outil accessible depuis l'espace utilisateur lui permettant de contrôler le profilage. Nommé *perf*, ce programme utilise l'interface noyau pour réaliser des mesures soit en échantillonnage soit en comptage. Différentes commandes sont disponibles pour compter les évènements (*stat*, *record*, *top*, *bench*) et afficher les résultats (*report*, *annotate*). Grâce à la commande *perf*, il est possible de compter les évènements en utilisant directement leur nom. La commande suivante peut être utilisée pour mesurer le nombre de transactions en lecture réalisées par un contrôleur mémoire. Il faut pour cela, vérifier que le noyau utilisé est capable d'accéder aux PMU uncore (première commande ci-dessous). Ensuite, il faut vérifier que le noyau supporte l'utilisation des noms d'évènements symboliques (deuxième commande ou `perf list`). Enfin, la troisième commande permet de compter le nombre d'évènements.

```
# ls /sys/bus/event_source/devices/* | grep uncore_imc
uncore_imc_0 uncore_imc_1 uncore_imc_3 uncore_imc_4 uncore_imc_5

# ls /sys/bus/event_source/devices/uncore_imc_1/events/
cas_count_read  cas_count_read.scale  cas_count_read.unit clockticks ...

# perf stat -a -e uncore_imc_0/cas_count_read/
```

Malheureusement, lorsque les évènements voulus ne sont pas supportés par le noyau, il est nécessaire de se reporter à la documentation de l'architecture. Perf (comme PAPI) est capable de configurer les compteurs grâce à leur encodage (numéro de l'évènement, masque de configuration). Dans notre exemple, l'évènement est le numéro $0x04$ associé au masque $0x03$. Lorsque l'évènement est supporté par le noyau, c'est cette valeur qui est stockée dans les fichiers listés ci-dessus (première commande). Pour réaliser le même comptage que précédemment, la deuxième commande ci-dessous peut être utilisée :

```
# cat /sys/bus/event_source/devices/uncore_imc_1/events/cas_count_read
event=0x04,umask=0x03

#perf stat -a -e "uncore_imc_0/event=0x04,umask=0x03/"
Performance counter stats for 'system wide':
4.94 MiB  uncore_imc_0/cas_count_read/
```

Le travail de Andy Kleen (*pmu-tools*[42]) peut aussi être utilisé pour faciliter l'utilisation de `perf` lorsque les noms symboliques des évènements ne sont pas supportés. `Ocperf` est un *wrapper* de la commande `perf` qui traduit les événements de la liste complète des événements Intel au format *perf* lorsque ceux-ci ne sont pas supportés (première commande) :

```
#perf  stat -e FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE sleep 1
invalid or unsupported event

#./ocperf.py  stat -e FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE sleep 1
perf stat -e cpu/event=0xc7,umask=0x4,name=fp_arith_inst_retired_128b_packed_double/ sl
... correct ...
```

Oprofile [LE04]. Oprofile [LE04] est un outil de suivi de performance développé par John Levon en 2001 dans le cadre de son projet de master et fut le principal outil de suivi de performance de Linux pendant plusieurs années[43]. Oprofile est capable d'utiliser des compteurs de performances matérielles pour suivre la performance des processus, des bibliothèques partagées et du noyau. Pour suivre ces performances, Oprofile utilise un démon permettant à l'utilisateur de spécifier un événement matériel à surveiller et un seuil d'événement pour déclencher l'interruption (échantillonnage). En utilisant les tables de symboles de débogage, il peut faire le lien entre les adresses des instructions et les lignes du code source associées. Oprofile est un outil mature possédant une multitude d'options telle que la génération de graphiques d'appel (*call graph*). Plusieurs utilitaires sont fournis à l'utilisateur pour contrôler le suivi de performance (*opcontrol*, *opreport* ...). À l'origine, Oprofile utilisait un module noyau nécessaire pour accéder aux PMU. Quand *Perf Events* fut introduit avec son interface noyau, Oprofile a alors été adapté pour l'utiliser lui aussi. La communauté autour de la commande `perf` est sans doute plus active et dynamique, et de nombreuses nouvelles fonctionnalités sont ajoutées à *perf* sans analogies dans *OProfile*.

PAPI [Bro+00]. L'interface Performance Application Programming Interface (PAPI), a pour objectif de simplifier l'utilisation des PMU de différentes architectures. Pour cela, PAPI offre une abstraction pour un grand nombre de compteurs d'évènements pour le développement d'outils de suivi de performance. Pour accéder aux MSR, PAPI utilisait l'interface *perfctr* avant de basculer sur l'interface offerte par *Perf Events* avec la version 2.6.32 du noyau Linux. PAPI supporte le comptage, l'échantillonnage ainsi que le multiplexage. Utiliser la totalité des fonctionnalités de PAPI nécessite une certaine expérience. Cependant pour un usage basique, son utilisation est très simple comme le montre l'exemple de l'extrait 2.3.

```
1 #define NUM_EVENTS 2
2 long_long values[NUM_EVENTS];
3 unsigned int Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};
4 PAPI_start_counters((int*)Events,NUM_EVENTS); /* Start the counters */
```

42. pmu-tools : https://github.com/andikleen/pmu-tools
43. Source : https://www.ibm.com/developerworks/linux/library/l-evaluatelinuxonpower/

```
5 do_work();
6 PAPI_stop_counters(values,NUM_EVENTS); /* Stop counters and store results*/
```

Extrait 2.3 – Utilisation simple de PAPI pour mesurer deux évènements dans un programme C.

PAPI est composé de deux parties permettant de compter des évènements dits *natifs* ou *prédéfinis* :

— Les événements **natifs** comprennent l'ensemble des événements qui peuvent être comptés par le CPU. Dans de nombreux cas, ces événements seront disponibles par le biais d'un événement PAPI prédéfini correspondant. Il y a généralement beaucoup plus d'événements natifs disponibles qu'il n'est possible d'en faire correspondre sur des événements prédéfinis du PAPI. Même si aucun événement prédéfini n'est disponible, les événements natifs sont toujours accessibles directement. L'utilisation de ces évènements nécessite d'avoir une bonne connaissance de l'architecture utilisée. Chaque évènement peut être configuré grâce à un *masque* pour désigner précisément l'évènement à compter de la même façon que lors de la programmation des MSR grâce aux commandes *rdmsr* et *wrmsr* présentées dans la section B.2.1.

— Les événements **prédéfinis** sont un ensemble commun d'événements jugés pertinents et utiles pour le réglage des performances des applications. Ces événements se trouvent généralement dans de nombreux CPU fournissant des compteurs de performance et donnent accès à la hiérarchie de la mémoire, aux événements du protocole de cohérence du cache, au nombre de cycles et d'instructions, à l'unité fonctionnelle et au statut du pipeline. En outre, les événements prédéfinis sont des correspondances de noms symboliques (nom prédéfini du PAPI) à des définitions spécifiques à la machine (événements natifs). Par exemple, le nombre total de cycles passés en mode utilisateur est PAPI_TOT_CYC [44]. Un évènement prédéfini peut utiliser une combinaison d'un ou plusieurs évènements natifs. Par exemple, l'évènement permettant de compléter le nombre de calculs flottants à simple précision réalisés, peut être mesuré avec l'événement prédéfini *PAPI_SP_OPS*. Cet évènement utilise en réalité quatre évènements natifs pour compter les instructions vectorielles de différentes tailles pour ensuite calculer le résultat *Number of FLOPS* (voir ci-dessous).

```
#papi_avail -e PAPI_SP_OPS
    Number of Native Events:      4
    Long Description:            |Single prec. op; vector|
    Postfix Processing String:   |N0|N1|4|*|+|N2|8|*|+|N3|16|*|+||
    Number of FLOPS= N0 + N1 * 4 + N2 * 8 + N3 * 16
```

Il existe une centaine d'évènements prédéfinis par PAPI. La disponibilité de ces derniers peut être vérifiée grâce à la commande `papi_avail`. Il est possible de créer ses propres jeux d'évènements prédéfinis. En raison des différences d'implémentation matérielle, il n'est pas

44. source :https://icl.cs.utk.edu/projects/papi/wiki/Events

toujours possible de comparer directement les comptes d'un événement prédéfini par PAPI obtenu sur différentes plates-formes matérielles.

Likwid [THW10]. *LIKWID* est un ensemble d'outils utilisables en ligne de commande pour aider les développeurs dans leur travail de caractérisation de plateformes et d'analyse de performances. Les outils fonctionnent sur la majorité des distributions Linux grâce à sa faible dépendance à des libraires externes : il ne nécessite que le compilateur *GNU compiler*. L'outil possède 11 outils pouvant être groupés en trois catégories : analyse de performance, caractérisation de plateforme et contrôle de l'exécution. Deux outils sont importants pour analyser les performances d'une application : *likwid-perfctr* et *likwid-perfscope*.

Le premier permet d'accéder aux compteurs de performances à travers les interfaces principales telles que *Perf Events* ou *perf_ ctr*. Les principaux évènements sont accessibles (*core* et *uncore*) et la majorité des processeurs x86 sont supportés grâce au sous-système *perf_ event_ open()* (Intel Xeon, Intel Xeon Phi, et AMD Zen). L'avantage de cet outil est de pouvoir utiliser des groupes d'évènements déjà existants tels que :

```
1 FLOPS_SP     Operation flottante simple precision MFlops/s
2 L3           Bande passante du cache L3 en MBytes/s
3 MEM          Bande passante memoire en MBytes/s
```

L'outil peut être utilisé pour mesurer la performance d'un système ou d'une application particulière utilisant la programmation multicoeur. Likwid propose une API permettant d'annoter le code pour ne mesurer la performance que de certaines régions de l'application :

```
1 LIKWID_MARKER_START(Compute);
2 <code>
3 LIKWID_MARKER_STOP(Compute);
```

Pour pouvoir fonctionner, l'application doit être compilée avec le drapeau de compilation `-DLIKWID_PERFMON`. L'avantage est de pouvoir facilement désactiver la mesure de performance et n'ajouter aucun coût supplémentaire à l'application lors d'une utilisation en production.

Le deuxième outil, *likwid-perfscope*, est un outil de visualisation des données mesurées par le premier. Il permet par exemple d'afficher l'évolution du trafic mémoire lors de l'exécution de l'application ou de la consommation énergétique. Malheureusement, cet outil n'est pas supporté pour les processeurs Skylake. Après avoir discuté avec le développeur de cette application, il semble que cet outil soit un "*jouet*" et que le développeur originel ne fasse plus partie du projet.

BSC Performance Tools Le centre de recherche du Barcelona Supercomputing Center développe depuis 1991 une suite d'outils permettant la caractérisation des applications. Cette suite d'outils est principalement composée de trois outils : *Extrae* [Rod], *Paraver* [Pil+95] et *Dymemas* [LGC97]. La philosophie d'utilisation de ces outils est d'aider l'utilisateur à trouver le goulot d'étranglement (bottleneck) des performances de son application qui peut résulter de plusieurs causes. Les outils ne sont pas une boîte noire répondant à toutes les questions automatiquement. La méthodologie d'analyse, comprenant l'utilisation de *Paraver* et *Dimemas*, est basée sur l'examen de la distribution temporelle et spatiale des données de performance pour

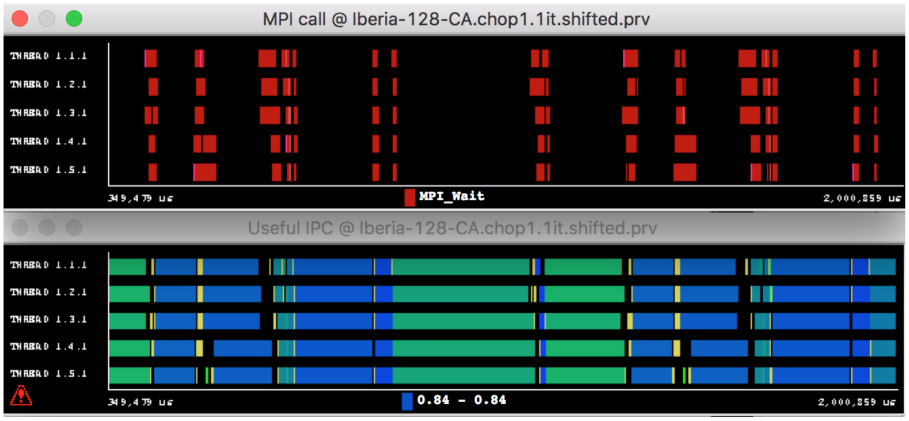


FIGURE 2.60 – L'outil *Paraver* permet d'afficher le graphique des traces obtenues grâce à l'outil *Extrae*.

comprendre le comportement de l'application, détecter ses différentes phases et identifier la structure comportementale (qui peut être différente de la structure procédurale). Cette analyse détaillée permet d'extraire beaucoup d'informations des données de performance recueillies au cours de l'exécution. L'utilisateur doit être un programmeur confirmé dans l'analyse de performance et utiliser les outils pour l'aider dans son analyse de performance. Les outils permettent de facilement comparer plusieurs traces correspondant à différentes exécutions de l'application (paramètres ou architectures différents).

L'outil *Extrae* (*extraire*). Il permet notamment d'instrumenter dynamiquement un code pour en extraire des traces qui pourront être visualisées avec l'outil *Paraver* (*pour voir*). L'outil injecte une librairie (`LD_PRELOAD`) pour annoter les différents symboles de l'application. La librairie *Dyninst* est utilisée pour réaliser l'instrumentation au niveau du code source ou directement dans le fichier binaire. L'ensemble des outils est compatible avec *OpenMP* et *MPI* pour permettre d'obtenir des profils d'exécutions parallèles. La figure 2.60 montre un exemple de deux traces dont les graphiques sont alignés pour faciliter l'extraction de résultats. La première fenêtre montre en rouge les appels à des fonctions de communication de la librairie *MPI*. Les parties en noir représentent le code de l'application propre à chaque coeur. La deuxième fenêtre montre à l'aide d'un dégradé de couleur le nombre d'instructions utiles par cycle. Les couleurs claires représentent un *IPC* plus faible. On remarque ainsi que les communications ne sont pas parfaitement synchrones et que des coeurs perdent un certain temps à attendre que les autres coeurs aient fini leur partie du travail. L'outil *Paraver* possède une visualisation sous forme de tableur permettant de quantifier le temps perdu à cause de ces désynchronisations et d'identifier le nom des fonctions responsables.

Le troisième outil, *Dimemas*, permet à l'utilisateur de développer et optimiser des applications parallèles sur son poste de travail, tout en fournissant une prédiction précise de leurs performances sur une architecture parallèle. Le simulateur Dimemas reconstruit le comportement temporel d'une application parallèle sur une machine modélisée par un ensemble de paramètres de performance. Ainsi, les expériences de performance peuvent être faites facilement. L'absence

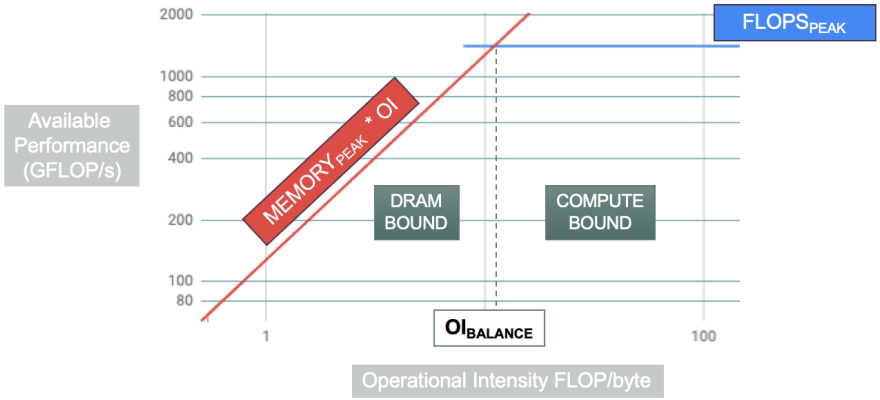


FIGURE 2.61 – Représentation graphique du modèle du *Roofline*. En fonction de son intensité opérationnelle, la performance d'un code sera limitée par la bande passante (memory bound) ou par le processeur (compute bound).

de latence du réseau (virtuel) permet d'estimer les performances de son application si elle utilisait un supercalculateur avec un réseau parfait. L'utilisateur peut ensuite estimer l'impact apporté par le portage sur MPI et OpenMP. Les traces générées par *Dimemas* peuvent ensuite être analysées avec l'outil *Paraver*.

2.4.2.5 Modèle de performance : le Roofline model

Présenté par William et al. en 2009 [PWW09], le modèle du *Roofline* est un modèle de performance simple qui représente graphiquement les performances d'un code en situant sa performance par rapport aux performances maximales de l'architecture. L'objectif principal de ce modèle est de donner le pourcentage de la performance disponible atteinte par un code. L'intérêt de ce modèle est de restreindre l'analyse de performance aux deux ressources importantes pour les applications HPC : la performance calculatoire (GFLOPS) et la performance du bus mémoire (GB/s). Le modèle est utilisé pour représenter les différentes fonctions clefs d'une application. Ainsi, le programmeur pourra commencer son travail d'optimisation sur les fonctions avec le plus de potentiel.

La figure 2.61 montre une représentation du modèle du *roofline*. Sur l'axe des abscisses est représentée l'intensité opérationnelle de l'algorithme (en FLOPS/byte) qui correspond au nombre d'opérations flottantes appliquées à chaque byte de donnée amené depuis la mémoire. Sur l'axe des ordonnées est représentée la performance de calcul mesurée en GFLOPS. Chaque hot spot est placé en fonction de son intensité opérationnelle, calculée à partir de la lecture du code, et de sa performance, mesurée lors de l'exécution.

Modélisation L'objectif du modèle est de déterminer si la performance du code pour une architecture donnée est structurellement limitée par la performance du processeur ($\texttt{FLOPS}_{\texttt{peak}}$ en $GFLOP/s$) ou bien par la performance du bus mémoire ($\texttt{MEMORY}_{\texttt{peak}}$ en GB/s). Une ap-

plication réalisant la lecture de deux nombres pour y réaliser des centaines d'opérations verra ses performances limitées par la capacité de calcul $\texttt{FLOPS}_{\texttt{peak}}$ du processeur. Inversement, une application devant lire un grand jeu de données pour ne réaliser qu'une opération sur chaque valeur verra ses performances limitées par celle du bus mémoire $\texttt{MEMORY}_{\texttt{peak}}$. On peut estimer la quantité de calculs à réaliser sur chaque donnée en calculant son Intensité Opérationnelle (OI en $FLOP/byte$). Pour cela il faut lire le code source pour compter manuellement le nombre d'opérations réalisées (#FLOP) et le nombre de données nécessaires chargées depuis la mémoire (#BYTE). On peut ainsi calculer l'Intensité Opérationnelle d'un code en faisant le ratio des deux valeurs :

$$\mathrm{OI}_{kernel} = \frac{\#\mathrm{FLOP}}{\#\mathrm{BYTE}} \tag{2.11}$$

Le temps pour exécuter le code ($\mathrm{TEMPS}_{theorique}$), sera le temps mis par la ressource la plus utilisée par le code. On peut estimer ce temps par la formule suivante.

$$\mathrm{TEMPS}_{theorique} = max \begin{cases} \dfrac{\#\mathrm{FLOP}}{\mathrm{FLOPS}_{peak}} \\ \\ \dfrac{\#\mathrm{BYTE}}{\mathrm{MEMORY}_{peak}} \end{cases} \tag{2.12}$$

La performance théorique du code ($\mathrm{PERF}_{theorique}$ en GFLOPS) peut être calculée grâce aux transformations successives de l'équation 2.13.

$$\frac{\mathrm{TEMPS}_{theorique}}{\#\mathrm{FLOP}} = \max \begin{cases} \dfrac{1}{\mathrm{FLOPS}_{peak}} \\ \\ \dfrac{\dfrac{\#\mathrm{BYTE}}{\mathrm{MEMORY}_{peak}}}{\#\mathrm{FLOP}} \end{cases}$$

$$\frac{\#\mathrm{FLOP}}{\mathrm{TEMPS}_{theorique}} = \min \begin{cases} \mathrm{FLOPS}_{peak} \\ \\ \dfrac{\#\mathrm{FLOP}}{\#\mathrm{BYTE}} \times \mathrm{MEMORY}_{peak} \end{cases} \tag{2.13}$$

$$\mathrm{PERF}_{theorique} = \min \begin{cases} \mathrm{FLOPS}_{peak} \\ \\ \mathrm{OI}_{kernel} \times \mathrm{MEMORY}_{peak} \end{cases}$$

Pour une architecture, il faut déterminer pour quelle intensité opérationnelle une application

est limitée par la mémoire ou le processeur. Pour cela, il faut calculer l'intensité opérationnelle ($\mathrm{OI}_{balance}$) correspondant au croisement des deux droites sur la figure 2.61.

$$\begin{aligned} \mathrm{FLOPS}_{peak} &= \mathrm{OI}_{balance} \times \mathrm{MEMORY}_{peak} \\ \mathrm{OI}_{balance} &= \frac{\mathrm{MEMORY}_{peak}}{\mathrm{FLOPS}_{peak}} \end{aligned} \tag{2.14}$$

Une application dont l'intensité opérationnelle est inférieure à $\mathrm{OI}_{balance}$ verra sa performance limitée par le système mémoire. Plus rarement, si l'intensité opérationnelle d'une application est supérieure à $\mathrm{OI}_{balance}$, la performance sera alors limitée par le processeur.

Construction La première étape dans la construction du graphique est de tracer les deux axes limitant les performances d'un code. Ces deux droites représentent les performances crêtes de la mémoire et du processeur. Pour obtenir ces valeurs, elles peuvent être calculées à partir des spécifications du processeur. Cependant, avec la complexification des architectures, il est difficile de les atteindre même avec des benchmarks prévus à cet effet. Il est donc préférable de les représenter par des valeurs mesurées comme indiqué dans la littérature [Far14]. Pour la mémoire, le benchmark STREAM peut être utilisé. Pour la performance du processeur, nous utilisons le générateur de benchmarks présenté dans la section 3.3. D'autres travaux sont venus compléter les benchmarks disponibles pour caractériser l'architecture [Lo+15].

Évolutions Le *roofine* a reçu de nombreuses améliorations depuis sa création. En 2014, les travaux [IPS14] constatent que le modèle original n'est pas suffisamment précis à cause de la faible précision de caractérisation de l'architecture. En effet, un code pouvant profiter de la localité des données dans les caches pourrait atteindre des performances supérieures au maximum prévu par le modèle utilisant seulement la bande passante mémoire. Inversement, la performance crête est calculée pour un code utilisant tous les coeurs du processeur, avec des instructions FMA vectorisées. Cependant, par leur nature, certains codes ne peuvent pas utiliser ces caractéristiques. La performance crête étant alors impossible à atteindre. Le modèle Cache-Aware Roofline Model (CARM) [IPS14] a ainsi été développé permettant de représenter la performance des différents niveaux de caches. Cependant, le programmeur doit comprendre si son application peut profiter de cette localité, ce qui peut rendre cette approche plus difficile. Le modèle a depuis été affiné avec le Locality Aware Roofline Model (LARM) [Den+18] permettant de modéliser les accès en mémoire non uniforme (NUMA). D'autres travaux essaient d'automatiser sa construction [Lo+15] pour faciliter son usage. L'outil de profiling d'Intel a intégré les modèles CARM et LARM pour automatiser la recherche des hot spot et afficher leur performance sur un même graphique. Pour cela, il désassemble le code et calcule l'intensité opérationnelle de la boucle étudiée.

Critiques La force de cette approche est de montrer rapidement au programmeur si son application est efficace ou non. Dans le cas échéant, il sait s'il doit travailler sur l'optimisation des FLOPS ou de la mémoire. En modélisant les principaux noyaux de calcul (kernels)s de son application, le programmeur saura sur lesquels ses optimisations seront les plus bénéfiques.

Bien qu'ayant reçu de nombreuses améliorations, ce modèle doit être utilisé pour commencer l'analyse de performance. Cependant, il ne permet pas de modéliser ni de comprendre finement la raison d'une performance. La majorité des applications étant limitée par la bande passante mémoire, il est rare d'utiliser ce modèle pour modéliser la performance des unités de calculs. Mais il peut être intéressant de calculer l'intensité opérationnelle d'une boucle pour s'en assurer avant d'apporter des optimisations. De plus, les accélérateurs à venir essaient de réduire le trou de performance entre les processeurs et la mémoire. Cette modélisation est donc importante pour l'analyse de performance.

2.5 Conclusion

Ce chapitre a permis de réaliser une large étude du domaine du HPC et de comprendre les principaux défis à relever pour développer les prochaines générations de supercalculateurs.

Dans la section 2.1, nous avons commencé par rappeler les origines du calcul haute performance en présentant les simulations numériques. Les applications développées dans ce domaine nécessitent de grandes puissances de calcul qu'elles peuvent obtenir en accédant à d'immenses plateformes, appelées supercalculateur. Nous avons ensuite étudié les différents paradigmes de programmation parallèle permettant aux programmeurs d'accéder aux milliers de ressources de calculs disponibles. Cette section nous a permis d'introduire les concepts de performance en étudiant notamment la scalabilité des applications à l'aide des lois d'Amdahl et de Guftafson.

Dans la section 2.2, nous nous sommes intéressés à l'étude du classement du Top500. Ce classement réalisé depuis 1993 nous a permis de comprendre les tendances de l'évolution des performances des supercalculateurs. Nous avons ainsi discuté des principaux freins technologiques qui ont donné lieu à un ralentissement de l'évolution des performances au début des années 2010 (lois de Moore et de Dennard) ainsi que du déséquilibre des performances du système mémoire et des capacités de calcul des processeurs. Nous avons ensuite discuté de la nécessité de développer des plateformes plus puissantes permettant d'analyser le tsunami de données produit par les objets connectés, de prendre des décisions plus complexes (intelligence artificielle, simulations numériques précises) et plus rapides. Cette nouvelle génération de supercalculateurs nommés exascale permettra d'atteindre une puissance de calcul dix fois plus grande que celles atteintes par les plateformes actuelles. Pour cela, nous avons discuté des 6 principaux défis auxquels doit faire face l'industrie du HPC, dont celui de l'énergie. Les 10 premiers supercalculateurs du Top500 consomment entre 7 et 20 MW alors que l'objectif est de construire un supercalculateur dix fois plus puissant consommant entre 20 et 30 MW. Ces contraintes nous obligent à utiliser de nouvelles technologies très différentes de celles utilisées

actuellement, mais aussi de repenser en profondeur l'architecture des plateformes.

Dans la section 2.3, nous avons présenté les principales opportunités disponibles pour répondre aux défis précédemment évoqués. Pour faire face aux contraintes énergétiques et économiques, des technologies de ruptures sont actuellement développées telles que les mémoires SCM *Storage Class Memory*, les technologies photoniques et le protocole GEN-Z. Grâce à ce protocole, de nombreuses technologies pourront être utilisées pour exécuter les différentes fonctions d'une application de façon optimale. Afin de pouvoir profiter de l'hétérogénéité des solutions disponibles, il est nécessaire de caractériser chacune d'entre elles et d'adapter les applications pour en extraire le maximum de performance.

Enfin, dans la section 2.4 nous avons présenté les différentes méthodes et outils permettant de caractériser et de suivre les performances des architectures. Nous avons présenté plusieurs des nombreux benchmarks existants qui permettent de caractériser de nombreuses parties de la microarchitecture. Cette étude nous a permis de relever le manque de deux benchmarks permettant la caractérisation des unités arithmétiques et logiques et du système mémoire :

— Pour exécuter les instructions sur des nombres à virgule flottante, les ALU possèdent une unité de calcul spécialisée appelée FPU (*floating point unit*). Sur un processeur moderne, la FPU est capable d'exécuter plusieurs instructions vectorielles par cycle (entre 1 et 4). L'expérience nous a montré que l'anticipation de ses performances était très difficile. Certaines particularités de cette unité (gestion des dépendances, exécution dans le désordre, fréquence soutenable) doivent être caractérisées pour aider à la compréhension de la performance d'une application.

— Les applications HPC utilisent couramment des accès mémoires réalisant des sauts entre deux adresses mémoires. Ces sauts (appelé *stride*) peuvent être rencontrés dans de nombreux algorithmes. Par exemple, pour accéder à un champ appartenant à des objets stockés dans un tableau. L'adresse du champ des différents objets (dont la taille est constante) est séparée par un espace constant. L'accès au champ voulu des différents objets réalise donc des accès mémoire suivant un motif de saut. D'autres applications comme le calcul matriciel parcourent des matrices et génèrent des accès mémoire par saut de taille constante (taille d'une ligne de la matrice). Pour des tailles de sauts suffisamment grandes, l'accès suivant n'est pas présent dans la ligne de cache transférée lors de l'accès précédent. Afin d'obtenir des performances décentes, le pré chargeur de mémoire anticipe ces accès. La performance des applications dépend donc essentiellement de la capacité de ce matériel à comprendre le motif d'accès utilisé et d'anticiper les requêtes mémoires. Cependant, lorsque plusieurs jeux de données sont accédés simultanément, le préchargeur peut avoir des difficultés à les analyser. Ceci pouvant conduire à des performances désastreuses pour certaines applications clefs utilisées dans le HPC (algorithmes de stencil [Dat+08]) Nous remarquons qu'aucun outil précédemment étudié ne permet d'évaluer cette fonctionnalité critique du matériel.

Dans la fin de la section 2.4, nous avons présenté les différentes méthodes permettant d'ac-

céder aux compteurs matériels. Ces compteurs permettent de suivre l'activité de la microarchitecture et d'aider à l'analyse de la performance d'une application. Nous avons pu discuter de la difficulté de programmer et des nombreux inconvénients qui accompagnaient leur utilisation. Les compteurs matériels n'ont en effet que peu évolué ces dernières années. Les évolutions de performance des processeurs ne nécessitaient pas d'investir des ressources et du temps dans l'analyse des performances. Les architectures se complexifiant de plus en plus, il est aujourd'hui très difficile d'expliquer la mauvaise performance d'un code. Le manque d'outil disponible et le besoin d'expertise rendent ce travail d'analyse très difficile. Afin d'accompagner le programmeur dans ce travail, nous proposons de contribuer à l'aide de deux développements :

— Le système mémoire est la ressource critique des architectures actuelles. Sa bonne utilisation étant primordiale, il est nécessaire de posséder un outil simple d'utilisation permettant de suivre son activité.
— Afin de réaliser les optimisations les plus efficaces, il est nécessaire d'identifier les *hot spots* d'une application. Il est alors nécessaire de disposer d'un outil permettant d'extraire ces zones de codes et de proposer une analyse simple au programmeur.

Chapitre 3

Développements logiciels

Dans la partie précédente, nous avons présenté les évolutions majeures des plateformes utilisées dans le calcul haute performance. Nous avons évoqué les différentes opportunités technologiques et notamment l'utilisation de nouvelles architectures optimisées, rendue possible grâce au protocole `Gen-Z`. Nous avons discuté des principaux défis qu'il est nécessaire de relever pour pouvoir profiter de cette opportunité. Pour y répondre, nous avons développé quatre outils que nous présentons dans ce chapitre. Ces outils permettent de caractériser ces nouvelles microarchitectures et d'analyser la performance des applications exécutées.

Sommaire

3.1 Introduction . . . 114
- 3.1.1 Contributions . . . 114
- 3.1.2 Critères de développement . . . 115
- 3.1.3 Organisation du chapitre . . . 117

3.2 Benchmark mémoire . . . 118
- 3.2.1 Motivations . . . 118
- 3.2.2 Le benchmark DML_MEM . . . 120
- 3.2.3 Expérimentations et principaux résultats . . . 123
- 3.2.4 Conclusion . . . 141

3.3 Benchmark d'unité arithmétique . . . 143
- 3.3.1 Motivation et objectifs . . . 143
- 3.3.2 Kernel Generator . . . 145
- 3.3.3 Résultats . . . 156
- 3.3.4 Conclusion . . . 165

3.4 Monitoring du bus mémoire . . . 166
- 3.4.1 Introduction . . . 166
- 3.4.2 Développement de YAMB . . . 168
- 3.4.3 Utilisation . . . 170

3.5 Profil de l'exécution d'instructions . . . 172
- 3.5.1 Introduction . . . 173
- 3.5.2 Développement . . . 174
- 3.5.3 Résultats . . . 176
- 3.5.4 Conclusion . . . 180

3.6 Conclusion . . . 182

3.1 Introduction

Les nouvelles technologies présentées dans la section 2.3 vont permettre le développement d'une multitude de nouvelles architectures très différentes de celles utilisées aujourd'hui. Grâce à un protocole tel que Gen-Z (voir section 2.3.5), de nombreuses technologies pourront être agrégées pour exécuter les différents noyaux d'une application de façon optimale. La principale piste pour la construction d'une plateforme Exascale est d'utiliser ces architectures ultra-optimisées pour certains noyaux de calculs exécutés.

Dans le chapitre précédent, nous avons expliqué qu'une contrainte majeure dans le développement des nouvelles générations de supercalculateurs venait de l'énergie. L'amélioration de la performance énergétique ne viendra pas seulement des architectures, mais aussi de la capacité des codes à les utiliser correctement, ce qui est loin d'être le cas aujourd'hui. La plupart des supercalculateurs atteignent rarement 80% d'efficacité sur une application simple comme le benchmark HPL [DLP03], peu représentative des applications industrielles. Pour des applications réelles, cette efficacité est encore plus faible, parfois inférieure à 10% [Oli+05]. La principale source de consommation d'énergie ne provient pas du calcul, mais du déplacement des données [Kot16], l'optimisation de ces transferts est donc cruciale. De plus, comme la performance de la plupart des applications scientifiques est limitée par la bande passante mémoire, le travail de compréhension et d'optimisation est naturellement devenu la préoccupation de nombreux chercheurs et ingénieurs [McC95].

3.1.1 Contributions

L'hétérogénéité des centres de calculs est l'unique solution pour l'élaboration de plateformes Exascale. Celle-ci sera présente à plusieurs niveaux du supercalculateur : coeurs, processeurs, technologies mémoire... (voir section 2.3.4.3). Cependant, cela nécessite d'avoir recours à de nouvelles architectures très différentes de celles utilisées aujourd'hui (x86, GPU). Ces architectures sont spécialisées pour exécuter efficacement certaines catégories d'algorithmes. Ainsi, elles doivent être caractérisées pour exécuter les applications (ou certaines fonctions) qui peuvent profiter de cette efficacité. Une fois les plateformes et les noyaux de calculs caractérisés, le programmeur peut estimer les coûts de portage d'une application. Cette mesure est à la fois un coût économique (prix de la plateforme, consommation électrique), mais aussi de temps pour réaliser la transformation de code nécessaire.

Pour réaliser ce travail de caractérisation et d'optimisation, le programmeur doit avoir à sa disposition différents outils. Une première famille composée de code de benchmark lui permet de caractériser une nouvelle architecture pour en connaître les principales caractéristiques et estimer son efficacité pour une application. Il doit ensuite utiliser un deuxième jeu d'outils pour obtenir le profil de son application. Chaque application HPC possède plusieurs zones de calculs intensifs qui peuvent avoir des caractéristiques différentes. Pour choisir la plateforme optimale pour l'un d'entre eux, le programmeur doit posséder un outil lui permettant d'extraire et de caractériser ces zones de codes.

Pour pouvoir bénéficier de l'hétérogénéité et de la multitude de nouvelles architectures disponibles, nous proposons une méthodologie. Celle-ci permet d'identifier et caractériser ces nouvelles plateformes, de modéliser la performance d'une application, pour enfin porter son code sur l'architecture choisie et en extraire la performance maximale. La méthodologie est résumée sur la figure 3.1 et présentée en détail dans le chapitre 4 :

1. Caractériser les plateformes : mesurer les performances crêtes et étudier les forces et les faiblesses de l'architecture
2. Modéliser la performance des noyaux de calcul (kernels)s
3. Choisir la plateforme adaptée aux besoins des kernels et réaliser les transformations du code nécessaire
4. Vérifier les performances atteintes, et optimiser le code pour approcher la performance crête.

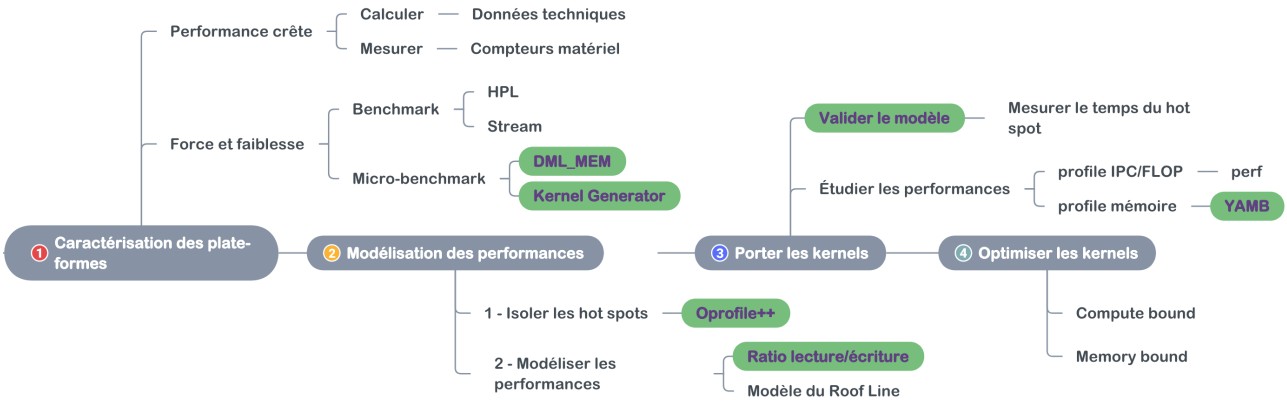


FIGURE 3.1 – Méthodologie pour caractériser et optimiser une application sur une nouvelle architecture. Les parties en vert représentent les développements présentés dans ce chapitre.

Pour chaque étape, nous avons sélectionné les outils nécessaires pour répondre aux questions ce qui permet de poursuivre la méthodologie. Pour cela, nous avons réalisé un état de l'art des outils existants, sélectionné ceux répondant à nos critères de développement et développé les outils manquants.

3.1.2 Critères de développement

Pour sélectionner et développer les outils nécessaires à ce travail, nous avons établi une liste de critères à respecter :

— **La simplicité** : notre principe de développement repose sur l'utilisation d'outils simples répondant à une question précise. En développant des outils simples, le programmeur peut facilement en comprendre le déroulement et se l'approprier en apportant les modifications qu'il estime judicieuses. En proposant les outils en source ouverte, nous espérons que l'expérience des différents utilisateurs puisse profiter au reste de la communauté. Bien

que certains travaux déplorent l'absence d'outils automatiques [Chu+12], nous estimons que la complexité des architectures et les multiples facteurs impactant la performance rendent impossible le développement d'outil automatique.

— **La compatibilité** : l'objectif de notre travail est la caractérisation de nouvelles plateformes, différentes de celles utilisées actuellement. Un critère majeur pour nos développements est d'assurer la compatibilité avec le maximum d'architectures. Le développement d'outils simples facilite d'autant plus la tâche du portage des outils. Ce critère interdit donc l'utilisation d'outils tels de `VTune`, utilisable uniquement pour les architectures Intel. Comme nous l'avons conclu dans la section 2.5, les compteurs matériels souffrent d'un manque de compatibilité entre les architectures (même d'un même constructeur) et peuvent être indisponibles sur des architectures de nouvelle génération. Les difficultés de programmation constatées dans l'Annexe B nous ont contraints à ne sélectionner et ne développer que des outils utilisant des compteurs matériels standards, ayant une grande chance d'être présents sur de nouvelles architectures. De plus, l'utilisation de compteurs plus complexes ne permet pas toujours de conclure facilement de la bonne ou mauvaise performance d'un code. Par exemple, un grand nombre de défaut de cache (miss) dans le cache de niveau 1 n'implique pas forcément une mauvaise performance s'il s'agit d'un algorithme de *stream*. Pour ces deux raisons, nous avons choisi d'utiliser et de développer des outils se basant uniquement sur des évènements simples :
  — nombre de cycles ;
  — nombre d'instructions exécutées ;
  — instructions mémoire en cours ;
  — nombre de miss dans le dernier niveau de cache.

— **La facilité d'utilisation** : le travail de caractérisation est une tâche très complexe. Si l'utilisation des outils doit être facilitée, il ne faut pas négliger la facilité d'installation. Plusieurs retours d'expérience nous ont montré que de nombreux utilisateurs n'utilisent pas certains outils lorsqu'ils ne parviennent pas à l'installer au premier essai. La réduction des dépendances à des librairies externes, et le développement d'outils simples nous assurent de réduire les difficultés lors de l'installation. De plus, dans des environnements industriels, l'utilisateur n'a pas toujours la liberté d'installer toutes les librairies qu'il souhaite le conduisant à renoncer à l'installation d'un outil. Les programmeurs qui travaillent quotidiennement sur des supercalculateurs savent que l'installation d'un outil, même simple, peut prendre du temps et de l'énergie : qu'il s'agisse de scripts de compilation complexes à déboguer ou de la dépendance vis-à-vis d'anciennes versions de bibliothèque. Pour ce faire, nos outils développés sont basés sur le nombre minimum de bibliothèques et l'outil de gestion de compilation libre de droits *cmake*[1]. CMake fait

1. CMake est un système de construction logicielle multiplateforme. Il permet de vérifier les prérequis nécessaires à la construction, de déterminer les dépendances entre les différents composants d'un projet (wikipédia) - https://cmake.org/

automatiquement la découverte et la configuration de la chaîne d'outils ce qui augmente la **portabilité** du code. Dans un environnement industriel, l'utilisateur n'aura pas non plus accès à des droits supplémentaires (`root` ou `kernel`). Nous avons donc sélectionné et développé des outils ne nécessitant pas ces droits. De plus, les compteurs matériels nécessitent des droits privilégiés, empêchant l'utilisation de nombreux outils existants. Cette raison est la principale motivation de l'utilisation du *backend* de `Perf Events` qui peut être rendu accessible à n'importe quel utilisateur. Bien que cette commande doive être exécutée par l'utilisateur `root`, elle ne doit être exécutée qu'une seule fois à l'allumage de la machine :
`sudo sh -c 'echo 1 >/proc/sys/kernel/perf_event_paranoid'`

3.1.3 Organisation du chapitre

Dans la section 2.4 nous avons relevé le l'absence de plusieurs outils nécessaire pour accompagner l'utilisateur dans le travail de caractérisation d'architectures et d'analyse de performance d'applications. À notre connaissance, soit ils n'existent pas, soit ceux existants ne répondent pas aux critères de développements cités précédemment. Ce chapitre présente les 4 principaux outils développés lors de ce travail de thèse. Il suit la structure suivante :

— La section 3.2 présente un nouveau benchmark permettant de réaliser des accès par sauts (strides) en mémoire pour des jeux de données de taille variable. Ce genre d'accès est très répandu dans les applications de type RTM [2] et il est important de pouvoir caractériser la microarchitecture pour ces codes-là.

— La section 3.3 introduit un générateur de benchmarks permettant de caractériser finement les unités de calcul arithmétique (ALU). Les architectures peuvent avoir des performances inégales et la performance de chaque instruction vectorielle doit être vérifiée pour valider la performance d'une application réelle (nombre d'instruction par cycle, fréquence supportée, opération flottante par seconde).

— La section 3.4 introduit un outil permettant de suivre l'évolution du trafic du bus mémoire. La mémoire étant une ressource critique des architectures modernes, cet outil est essentiel pour poursuivre la caractérisation des applications.

— La section 3.5 présente un outil capable d'extraire le code assembleur des points chauds (hot spots) d'une application et de caractériser chaque instruction ainsi que d'établir le résumé de la performance des boucles critiques.

2. Reverse Time Migration (RTM) - https://www.cgg.com/data/1/rec_docs/1899_RTM.pdf

3.2 Benchmark mémoire

Cette section présente notre benchmark mémoire appelé *Demonstrate Memory Limit* ou `DML_MEM`. Cet outil permet de vérifier le bon comportement de la hiérarchie mémoire lors d'accès mémoire à un jeu de donnée par sauts de taille constante.

3.2.1 Motivations

La performance du système mémoire est un élément critique dans la performance des applications. Celle-ci peut fortement varier en fonction du type d'accès mémoire réalisé. L'industrie de la recherche pétrolière est un grand consommateur de calcul haute performance. Les applications utilisées utilisent des algorithmes de *stencil*[3] qui impliquent des accès mémoire réguliers non contigus par *sauts* aussi appelés strides. D'autres applications comme le calcul matriciel parcourent des matrices et génèrent des accès mémoire par saut de taille constante (multiple d'une taille d'une ligne). La figure 3.2 expose un exemple simple de tels accès. D'autres algorithmes peuvent réaliser ce genre d'accès par *strides* comme les multiplications de matrices, les transformées de Fourrier ou le parcours d'un tableau de structures pour accéder à certains champs. Si les objets sont stockés continûment en mémoire, l'accès à un même champ de chaque objet réalise en réalité des accès mémoire par saut de taille fixe.

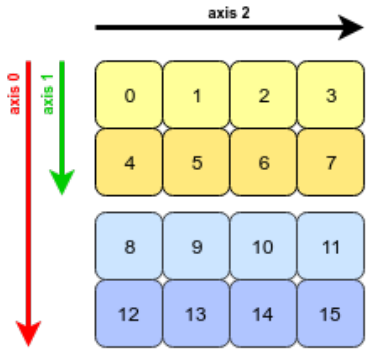
La grande majorité de ces applications ne réalisent pas suffisamment de calcul sur une donnée transférée pour masquer le temps de son accès mémoire. Ce déséquilibre de performance de l'architecture limite la performance de ces codes par celle du bus mémoire. Pour ces applications, il est primordial que l'architecture soit capable d'anticiper le maximum d'accès mémoire grâce à son prélecteur mémoire. Les prélecteurs mémoire sont conçus pour anticiper les accès avant qu'ils ne soient réalisés pour réduire la latence d'accès. Lorsque les accès sont simples (taille régulière, proches en mémoire), la majorité des architectures modernes obtiennent de très bonnes performances. Cependant, les accès par sauts peuvent être grands (supérieurs à plusieurs lignes de cache) et lorsque de multiples accès sont réalisés en concurrence, le prélecteur mémoire peut rencontrer des difficultés à les anticiper. De plus, si le prélecteur mémoire anticipe de mauvais accès, le bus mémoire sera saturé de données inutiles au calcul. Le bus mémoire étant la ressource critique pour la majorité des applications HPC, il est primordial que son utilisation soit la plus efficace possible.

3.2.1.1 Objectifs

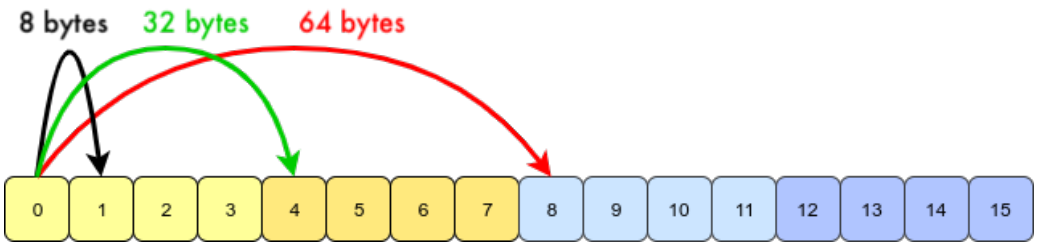
Le développement du benchmark `DML_MEM` a été réalisé pour répondre à trois objectifs :

1. **Caractériser la performance** d'une architecture pour l'exécution d'applications utilisant un motif d'accès mémoire par strides. En mesurant ses performances, il est ensuite

3. En mathématiques, en particulier dans le domaine de l'analyse numérique, un stencil est un arrangement géométrique d'un réseau nodal qui se lie au point d'intérêt en utilisant une routine d'approximation numérique. Les stencils sont à la base de nombreux algorithmes de résolution numérique des équations aux dérivées partielles (Wikipédia).



(a) Accès en ligne ou en colonne à une matrice.



(b) Les accès en ligne ou en colonne impliquent des sauts en mémoire de tailles différentes.

FIGURE 3.2 – Exemple d'une application réalisant des accès en colonne à une matrice. Ces accès impliquent en réalité des sauts entre les adresses mémoire utilisées.

possible de prouver l'efficacité de l'utilisation du sous-système mémoire pour une application réelle utilisant ce type d'accès. En effet, nous montrons que pour vérifier la bonne performance d'une architecture pour une application donnée, il n'est pas suffisant de vérifier que le bus mémoire est saturé.

2. **Attirer l'attention du programmeur** sur la complexité des architectures et de son impact sur les performances d'un code. Le benchmark développé doit pouvoir être utilisé pour caractériser l'ensemble de la hiérarchie mémoire : fonctionnement des caches et du prélecteur mémoire, saturation du bus mémoire, impact de l'utilisation de plusieurs coeurs, etc. En appréhendant cette complexité, il sera plus simple pour le programmeur d'apporter les bonnes modifications à son code pour tirer la pleine performance du bus mémoire.

3. **Aider à la conception** de nouvelles architectures en utilisant le benchmark pour vérifier le bon fonctionnement du système mémoire. En effet, en utilisant ce benchmark, nous avons trouvé plusieurs dysfonctionnements majeurs dans un accélérateur prévu pour améliorer l'exécution d'applications réalisant des accès mémoire par strides. Grâce à notre outil, nous avons pu prouver que les performances théoriques de la plateforme n'étaient pas accessibles par l'application.

3.2.1.2 Comparaison avec l'existant

L'étude des différents benchmarks existants est réalisée dans la section 2.4. À notre connaissance, il n'existe aucun benchmark permettant de caractériser l'architecture pour ce type d'accès. Le benchmark s'approchant le plus de cet objectif est celui de Saavedra [SS95]. Il utilise une taille de saut fixée au début de l'exécution pour accéder à un jeu de données. Cependant, la taille de ce dernier doit être un multiple d'une puissance de 2 et ne permet pas de dépasser la taille du dernier niveau de cache. Comme le souligne [YPS05], le problème d'une telle approche est de vouloir mesurer tous les niveaux de la hiérarchie simultanément. Les mesures peuvent alors être influencées par certains paramètres des différents niveaux de caches. Ces mesures doivent être interprétées par l'utilisateur, le programme ne créant pas automatiquement la hiérarchie. Un second outil s'approchant de notre démarche est le benchmark `DISBench` [FG13]. Cependant, il ne bénéficie d'aucune méthode solide de vérification de la performance comme celle implémentée par `DML_MEM`. De plus, il n'est, en aucun cas, prévu pour faciliter le test de multiples tailles de stride sur différentes tailles de jeux de données. Le code n'est plus maintenu depuis 6 ans et ne peut pas être exécuté sans erreur lors de l'exécution.

3.2.2 Le benchmark DML_MEM

Le motif d'accès par stride est donc très courant dans le calcul haute performance. La distance entre deux accès peut varier d'une application, ou d'un jeu de données, à l'autre. Pour caractériser les plateformes pour ces applications, il est donc nécessaire de posséder un benchmark paramétrable permettant de faire varier la taille du jeu de données et la taille du saut. Cette section présente comment le benchmark `DML_MEM` a été développé. Grâce à de nombreuses options, nous montrons comment différentes parties de la microarchitecture peuvent être testées : caches, répertoire de pages actives (TLB), bus mémoire.

3.2.2.1 Concept

Le principe du benchmark est d'accéder à un tableau en utilisant différentes tailles de strides. Pour chaque stride une mesure de performance est réalisée. Une fois toutes les tailles de stride mesurée, le benchmark augmente la taille du jeu de données utilisé (voir figure 3.3). La vitesse d'évolution de la taille des strides et du jeu de données peut être paramétrée. Les accès peuvent être réalisés en lecture ou en lecture/écriture.

Le benchmark se déroule en deux étapes : la configuration et son exécution. Lors de la configuration, les différents paramètres nécessaires pour l'exécution du benchmark sont extraits de la ligne de commande ou utilisent, le cas échéant, des valeurs par défaut. Les différentes versions du benchmark (mode d'accès, taille du déroulement des boucles) sont toutes compilées, l'initialisation utilise un pointeur de fonction vers la bonne version requise par l'utilisateur. Ensuite, le jeu de données est initialisé (voir section 3.2.2.3). La deuxième étape consiste à exécuter le benchmark et à mesurer ses performances. Pour une taille de jeu de donnée et une taille de stride, la bande passante effective maximale, minimale ou moyenne est mesurée et

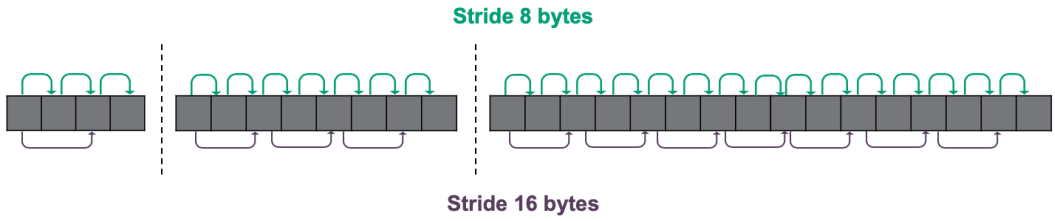


FIGURE 3.3 – Évaluation de la performance de deux tailles de stride (8 et 16 bytes) sur trois jeux de données de tailles différentes.

affichée. Pour cela, le benchmark mesure le temps nécessaire pour réaliser le noyau de calcul. Celui-ci, renvoie le nombre d'accès réalisés grâce à l'initialisation du tableau durant la première étape.

```
time1 = get_micros();
num_ops = p->m_BENCHMARK();
time2 = get_micros();
bande_passante = calcul_bw(num_ops, time2 - time1);
```

Ces mesures sont affichées au fur et à mesure de l'avancée du benchmark ainsi que dans un fichier de *log*. Ce fichier peut ensuite être utilisé par un script pour afficher l'évolution de la performance de chaque stride en fonction de la taille du jeu de donnée.

L'extrait présenté ci-dessous montre une exécution de l'outil. Pour trois strides de taille 8, 64 et 256 bytes, la performance en lecture est mesurée 10 fois. Pour différentes tailles de jeu de données, le débit moyen est affiché pour chaque taille de stride.

```
>./dml --type read --cacheline 64 --matrixsize 10000 --stride 8,64,256
...
Stride  S   ->          8         64        256
Value       ->    AVERAGE    AVERAGE    AVERAGE
       7.0 KiB     117.46          -          -
      81.0 KiB     114.29      88.24      74.12
      76.3 MiB      80.87      13.41       8.61
       7.5 GiB      80.35      13.11       7.18
...
```

3.2.2.2 Les options

Le benchmark `DML_MEM` accepte de nombreuses options. Grâce à celles-ci, différentes configurations peuvent être utilisées et permettent tester différents scénarios sur différentes parties de la microarchitecture. Nous décrivons ici les options les plus utiles pour l'utilisateur. Les nombreuses options peuvent être affichées avec l'option `--help`.

--stride, --minstride, --maxstride, --stridemode Ces quatre premières options permettent de définir quelles sont les différentes strides à utiliser pour réaliser les mesures. La première d'entre elles, permet à l'utilisateur de choisir une ou plusieurs à réaliser. Pour cela, une liste de taille de strides séparées par des virgules doit être entrée. Les trois dernières options permettent de générer automatiquement des strides à utiliser dans un intervalle $[min, max]$. L'option `--stridemode` peut être utilisée avec les valeurs *even* et *odd* pour décaler les strides ainsi générées. Cela permet d'éviter des strides utilisant seulement des multiples de deux, pouvant être affectées par certaines caractéristiques de la microarchitecture (taille de la ligne de cache, capacité des caches...).

--matrixsize, --minlog, --maxlog, --steplog, --log Ces 5 options permettent de définir la taille des jeux de données à utiliser. Leur taille évolue plus ou moins rapidement en fonction de la valeur de `--steplog` jusqu'à atteindre la taille `maxlog` ou bien celle de la matrice donnée avec l'option `--matrixsize`. Si `--steplog` est utilisé avec la valeur 0, le benchmark réalise la mesure sur un seul jeu de données de la taille `matrixsize`. L'option `--log` permet de donner la même valeur à `min` et `max` pour ne réaliser la mesure que sur une taille de jeu de données. Couplée avec l'option `--stride`, l'option `--log` permet d'utiliser le benchmark pour n'accomplir qu'une seule mesure : un jeu de donnée, une stride.

Ces 9 premières options sont les options principales du benchmark. Elles permettent de réaliser une multitude de mesures : performance des niveaux de caches, performance de la mémoire, fiabilité du prélecteur mémoire, mesure de la taille d'une ligne de cache. Dans la section 3.2.3.3 nous montrons comment ces options peuvent être utilisées pour identifier des tailles de strides ayant de mauvaises performances.

--type, --unroll, --mode Ces trois options permettent de choisir le benchmark à utiliser. L'option `--type` permet de réaliser les accès en lecture ou en lecture/écriture. La deuxième option permet d'appliquer l'optimisation du déroulement de boucle dont les différentes versions (déroulement de 2 à 64 fois) ont été programmées manuellement. La troisième option permet de choisir le mode d'accès. Par exemple, l'utilisation de différents pointeurs pour réaliser plusieurs accès notamment lorsque l'option `--unroll` est utilisée. L'analyse de la performance de ces options est réalisée dans la section 3.2.3.7.

--hugepages Cette option permet d'allouer la mémoire pour le jeu de données en utilisant des pages de 2 MiB contre 4 KiB habituellement. Un exemple de caractérisation des pages larges est présenté dans la section 3.2.3.8.

--annotate Cette option est utilisée lorsque l'activité du bus mémoire est mesurée avec l'outil `YAMB` présenté dans la section 3.4. Le benchmark annote le graphique lorsqu'une nouvelle taille de stride ou de jeu de données est utilisée. Grâce à cette option, il est possible de corréler l'activité du bus avec une configuration particulière du benchmark. Cette option est utilisée

dans la section 3.2.3.5 pour mesurer l'activité du bus mémoire lorsqu'un jeu de donnée de la taille du dernier niveau de cache est utilisé.

Version parallèle La dernière configuration du benchmark est la version parallèle utilisant MPI. Celle-ci doit être générée grâce à l'outil *cmake* et la commande `cmake -DOPT_BUILD_MPI=ON`. Grâce à cette version, plusieurs coeurs peuvent exécuter la même version du benchmark. Cette version du benchmark nous permet dans la section 3.2.3.4 d'étudier l'évolution du débit mémoire lorsque des coeurs supplémentaires sont utilisés.

3.2.2.3 Validation des résultats

Une grande difficulté lors de l'élaboration d'un benchmark a été de s'assurer que la performance mesurée était bien celle du code attendu. En effet, le compilateur peut appliquer certaines optimisations pour accélérer l'application. Ensuite, l'architecture elle-même peut se rendre compte de l'artificialité du code et en court-circuiter une partie. Dans les deux cas, le problème est que la mesure de la performance ne rend pas compte de la réalité du code et de la mesure attendue par le programmeur.

Pour éviter ces deux pièges, le benchmark `DML_MEM` initialise le jeu de données avec deux valeurs suivant le type de l'accès voulu (lecture ou lecture/écriture). Lorsque le benchmark utilise des accès en lecture, le jeu de données est initialisé avec la valeur **1**, car chaque lecture occasionne un transfert sur le bus mémoire. Lorsque le benchmark utilise le mode de lecture/écriture, le jeu de données est initialisé avec la valeur **2**. Chaque ligne doit être lue puis réécrite occasionnant deux passages sur le bus mémoire. Pour chaque accès, la valeur contenue dans le tableau est stockée dans une variable de compteur. L'utilisation de chaque valeur pour l'ajouter au compteur empêche le compilateur et l'architecture d'appliquer certaines optimisations. À la fin de l'exécution, le benchmark retourne cette variable permettant de compter le nombre total d'accès **effectivement** réalisés. En connaissant la taille du jeu de donnée utilisé ainsi que la taille des strides utilisée il est facile de valider cette valeur et de vérifier le bon fonctionnement de notre outil.

3.2.3 Expérimentations et principaux résultats

Dans cette section nous présentons les principaux résultats obtenus avec le benchmark `DML_MEM`. L'objectif est de montrer au lecteur les différentes mesures rendues possibles par l'utilisation de l'outil. Les tests sont principalement réalisés sur l'architecture des processeurs Intel Skylake. Les expérimentations suivantes sont présentées dans la suite de cette section :

- Mesure de l'impact du compilateur sur le débit mémoire ;
- Mesure de la taille d'une ligne de cache ;
- Évaluation de la performance de différentes tailles de stride ;
- Mesure du débit mémoire maximale atteignable ;
- Évaluation du fonctionnement des différents niveaux de cache ;
- Validation du fonctionnement des différents prélecteur mémoire ;

— Implémentation de l'optimisation du déroulement de boucle ;
— Évaluation de l'utilisation de pages larges ;
— Évaluation de l'impact de la fréquence des coeurs sur le débit mémoire.

3.2.3.1 Impact du choix du compilateur sur la performance

Contrairement au benchmark du générateur de kernel (voir section 3.3), le code de `DML_MEM` n'est pas écrit directement en assembleur. La qualité du compilateur peut donc avoir un impact significatif sur ses performances. Avant de réaliser plus d'expérimentations, nous avons testé deux compilateurs (GCC 8.2 et ICC 19.0) avec différents drapeaux de compilation. Avec d'anciennes versions de GCC (telle que la version 4.8), nous avons mesuré une amélioration d'un facteur deux en utilisant les drapeaux `-O3 -march=skylake-avx512`. Le compilateur ayant reçu de nombreuses améliorations depuis, nous n'avons trouvé aucun drapeau permettant d'améliorer les performances de ce dernier. Nous l'avons comparé avec la version 19.0 du compilateur d'Intel ICC couplé avec le drapeau `-O3`. Les performances mesurées dans les différents niveaux de la hiérarchie mémoire sont présentées dans le tableau 3.1.

Débit des différents niveaux mémoire (GB/s)	GCC 8.2	ICC 19.0
Débit L1 (GB/s)	58	310
Débit L2 (GB/s)	56	161
Débit L3 (GB/s)	26	26
Débit mémoire (GB/s)	12.5	12.5

Tableau 3.1 – Performance du benchmark `DML_MEM` configuré pour mesurer le débit maximal (GB/s) atteignable pour quatre niveaux de la hiérarchie mémoire. Le benchmark compare la performance atteignable lors de l'utilisation des compilateurs GCC et ICC. Le drapeau d'optimisation -O3 est utilisé dans les deux cas.

Lorsque le jeu de données tient dans le premier niveau de cache, nous avons mesuré des différences de performances du benchmark pouvant aller jusqu'à un facteur 8. Cet écart de performance entre les deux compilateurs se réduit lorsque la taille du jeu de données augmente. En effet, nous mesurons des performances équivalentes pour les deux versions de compilateurs lorsque le jeu de donnée accédé est localisé en mémoire. Nous expliquons l'écart de performance constaté dans les premiers niveaux de cache par la mauvaise performance du code généré par le compilateur GCC. Le premier niveau de cache des processeurs Skylake est capable de fournir 128 octets par cycle, soit une bande passante de 345 GB/s. Le code généré par GCC ne parvient pas à utiliser plus de 58 GB/s. Nous avons mesuré que le benchmark compilé par GCC utilise deux fois plus d'instructions que celui compilé par ICC. La performance du benchmark compilé par GCC n'est alors pas limitée par la performance du bus mémoire (memory bound) mais par celle du processeur (compute bound). La bande passante disponible se réduisant lorsqu'on "remonte" les niveaux de la hiérarchie mémoire, l'impact de la qualité du code est aussi réduit. Les

performances du benchmark compilé par ICC étant toujours supérieures à celles produites par GCC, nous utiliserons le compilateur d'Intel dans les prochaines expérimentations. Lorsque de nouvelles versions sont disponibles ou que d'autres architectures sont étudiées, nous conseillons de toujours tester les différentes versions de compilateurs avec les drapeaux de compilation adéquats. Ce constat réalisé sur notre benchmark est aussi applicable pour une application réelle.

3.2.3.2 Mesurer la taille d'une ligne de cache

Les transferts de données entre la mémoire et le processeur sont réalisés par paquet de données appelés *ligne de cache*. L'origine et les propriétés des caches sont présentées dans l'Annexe A.3.3. Connaître la taille d'une ligne de cache de l'architecture est nécessaire pour obtenir les mesures correctes par le benchmark. Cette taille peut aussi être nécessaire lors du développement d'une application pour disposer les données de façon optimale. Nous montrons dans cette expérimentation comment cette taille peut être retrouvée en utilisant le benchmark `DML_MEM`. Pour cela, nous désactivons le prélecteur mémoire (*memory prefetcher*) pour l'empêcher d'anticiper le chargement d'une ou plusieurs lignes de cache avant son accès. Le jeu de données utilisé doit quant à lui être plus grand que le dernier niveau de cache. La taille des lignes de cache des architectures modernes est généralement comprise entre 32 et 256 bytes. Nous utilisons le benchmark pour mesurer la performance du système mémoire en utilisant des strides de puissance de 2 allant de 8 à 256 bytes. Les performances ainsi mesurées sont présentées dans le tableau 3.2.

Taille de la stride (byte)	8	16	32	64	128	256
Bande passante (GB/s)	31.58	25.84	14.50	7.62	7.65	7.62

Tableau 3.2 – Performance de plusieurs tailles de stride pour un jeu de données de 1 GiB lorsque le prélecteur mémoire est désactivé.

L'interprétation de ces résultats est la suivante. Pour des strides de 64, 128 ou 256 bytes, la performance est la même. Il est important de rappeler que le benchmark mesure le débit mémoire atteint par l'application et non le trafic mémoire du bus. La performance de ces trois strides est égale. Peu importe la taille du saut réalisé, la donnée accédée lors du prochain accès sera sur une autre ligne de cache. Ceci indique que pour ces trois cas, le processeur attend une ligne de cache pour réaliser un accès. Lorsqu'une stride de 32 bytes est utilisée, la performance double, indiquant que la taille d'une ligne de cache est de 64 bytes. En effet, le processeur est capable de réaliser deux fois plus d'accès. Ceci est possible, car lorsqu'un premier accès est réalisé sur une ligne de cache, le suivant le sera aussi. La donnée est alors déjà présente dans le cache L1. L'utilisation d'une stride de 16 bytes améliore encore la performance du benchmark sans doubler pour autant. En effet, comme dans la première expérimentation le code devient compute bound. Le benchmark additionne des nombres flottants et la performance du code est alors limitée par l'ALU.

3.2.3.3 Performances de différentes tailles de strides

Un objectif principal de notre benchmark est de vérifier le bon comportement du processeur lors d'accès mémoire utilisant des sauts d'adresse de taille constante. Pour cela, nous avons développé un script qui permet d'exécuter le benchmark avec un grand nombre de strides et d'afficher leur performance dans un graphique. La figure 3.4 montre le résultat d'une telle exécution. Pour chaque stride et pour chaque taille de jeu de données, une mesure est réalisée. Pour faciliter la lecture du graphique, nous avons coloré les strides en fonction de leur taille en allant du bleu (pour les strides les plus petites) au rouge (pour les plus grandes). Nous remarquons que les strides de grande taille (plusieurs MiB) ont de meilleures performances que celle de petite taille. En effet, même pour des tailles de jeu de données de plusieurs centaines de megabytes (ne pouvant pas tenir dans le cache), le benchmark mesure des performances similaires celles mesurées lorsque le jeu de données se trouve dans le cache. En réalité, pour des grandes tailles de strides, le jeu de données réellement utilisé par le benchmark peut être contenu dans les différents niveaux de cache.

Nous remarquons sur la figure 3.4 que certaines strides ont des comportements différents que des strides de tailles proches (donc de couleurs proches aussi). En effet, des groupes de strides ont des performances bien plus faibles que d'autres et les mêmes résultats sont obtenus en utilisant des pages larges pour réduire l'impact sur la Translation Lookaside Buffer. Nous avons isolé certaines d'entre elles et reporté leur performance dans le tableau 3.3. Pour réaliser ces mesures, la commande suivante a été utilisée :

```
./dml --steplog 0.01 --unroll 8 --mode special --type read --cacheline 64
      --stride 73704,73728,77816,77824,81928,81920 --measure 10 --matrixsize 10000
```

Taille de la stride (byte)	Débit mémoire mesuré (GB/s)	Nb. inst.	IPC	LLC Miss
73704	24.54	690071400	0.34	21100474
73728	2.04	690064918	0.22	30612018
77816	24.53	688909428	0.33	21152403
77824	4.01	688905576	0.27	30144907
81928	24.76	690692156	0.33	21194483
81920	4.03	690693382	0.27	30794354

Tableau 3.3 – Mesure du débit mémoire atteint, du nombre d'instructions exécuté et le débit de leur exécution (Instruction Par Cycle) ainsi que le nombre de *miss* mesuré dans le dernier niveau de cache (LLC) pour trois couples de strides de taille similaire.

Le benchmark qui utilise une *mauvaise* stride (73728, 77824 ou 81920) voit sa performance limitée par la latence du système mémoire (*latency bound*). En effet, nous avons réalisé différentes mesures telles que le nombre de *miss* du dernier niveau de cache ou l'activité du bus

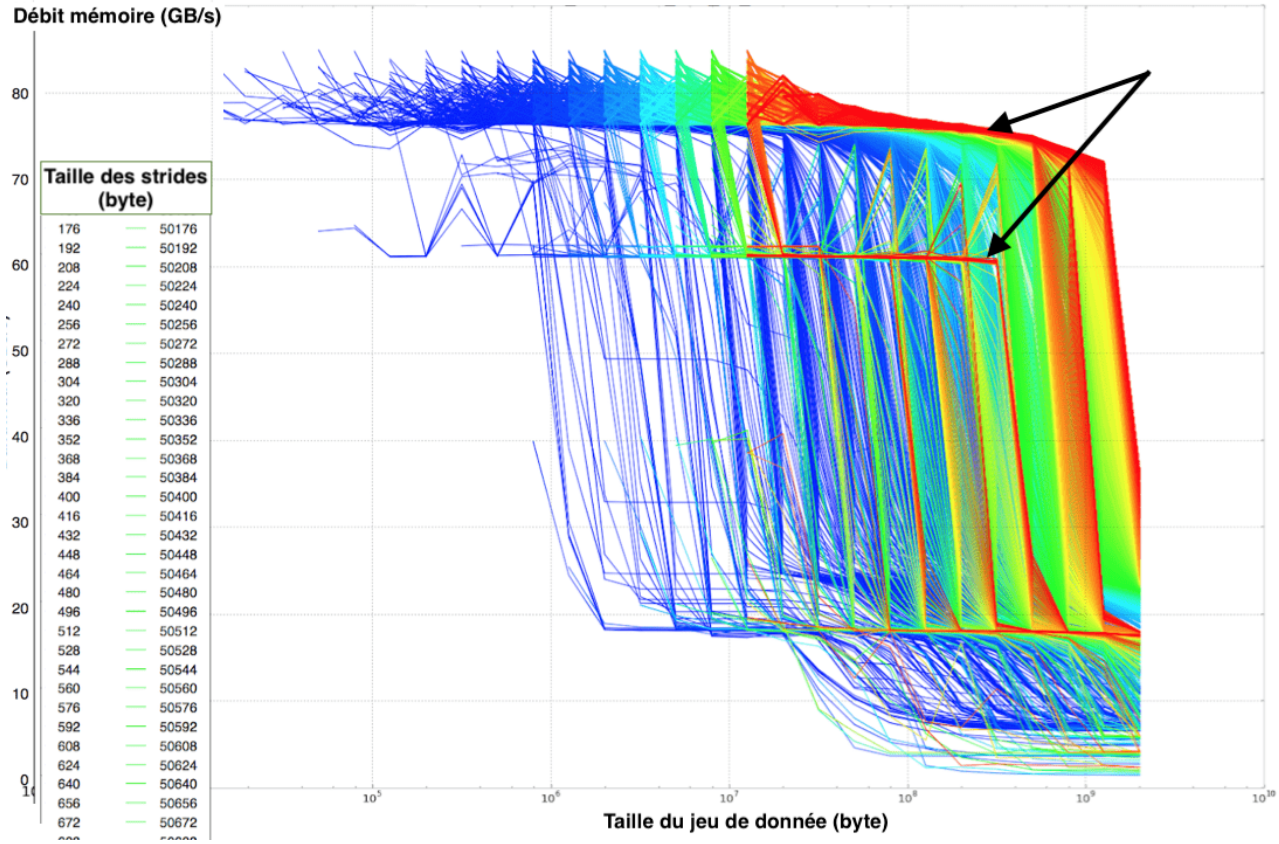


FIGURE 3.4 – Mesure du débit obtenu à l'aide du benchmark `DML_MEM`. En abscisse : taille du jeu de donnée en byte. En ordonnée : débit mesuré en GB/s. Les différentes tailles de strides sont représentées par des couleurs allant du bleu (petite taille) au rouge (grande taille). Nous remarquons que certaines strides de taille similaire (couleur proche) ont des performances très inégales (flèche).

mémoire. L'analyse de l'activité du bus mémoire montre qu'il est loin d'être saturé. Si la ligne de cache n'est pas présente dans un des niveaux de cache, et que le bus n'est pas saturé, c'est que le processeur l'attend et n'a pas anticipé son manque (*miss*). On remarque que le nombre d'instructions réalisées par cycle d'horloge (IPC) est plus faible et que le nombre de miss est lui plus élevé pour ces strides. La question est alors de savoir pourquoi pour une certaine taille la ligne de cache est présente dans le cache et que pour une stride plus grande de quelques bytes elle n'y est pas. L'explication vient de la taille des strides utilisées. Nous avons utilisé des strides de taille $Stride_{n+1} = Stride_n + 16\ bytes$ avec $Stride_0 = 16\ bytes$. En utilisant des tailles multiples de 16, certaines d'entre elles génèrent des conflits avec la politique de remplacement de lignes de caches. Ainsi, ces tailles de saut particulières mettent la pression seulement sur une partie du cache, le rendant inefficace. Le processeur doit donc attendre pour une majorité des accès que la ligne de cache soit transférée depuis la mémoire. La performance du code est alors limitée par la latence du système mémoire (*latency bound*).

Nous avons ensuite réalisé la même expérimentation en décalant la taille des strides utilisées en commençant avec une stride minimale $Stride_0 = 8\ bytes$. Ainsi, aucune taille de stride utilisée n'est multiple de 32, et aucune d'entre elles n'obtient de performance inattendue (voir figure 3.5).

À travers cette expérimentation, nous avons voulu montrer qu'un code simple peut avoir des performances inattendues. La complexité des architectures modernes est telle qu'elle peut avoir une incidence forte sur la performance des applications. Pour des strides aussi longues (plusieurs MiB), le prélecteur mémoire ne semble pas arriver à anticiper ces accès. Si une application réalise ce type d'accès, le programmeur doit s'assurer de ne pas réaliser des strides de cette taille en ajoutant du *padding* (remplissage) pour décaler artificiellement les données accédées. Une autre optimisation lors d'accès à certains champs d'objets contenus dans un tableau est de regrouper ces mêmes champs dans une structure spécifique pour être contigus en mémoire.

3.2.3.4 Saturation du bus mémoire

Pour pouvoir modéliser la performance des applications, il est courant d'utiliser les performances maximales atteignables par une ressource. C'est le cas de notre approche présentée dans le chapitre suivant ou celle du modèle du Roofline (voir section 2.4.2.5). Pour cela, nous avons exécuté le benchmark en utilisant différents nombres de coeurs. Les résultats sont visibles sur le graphique de la figure 3.6. Sur ce processeur, notre benchmark arrive à obtenir une bande passante mémoire maximale de 114 GB/s. À titre de comparaison, le benchmark STREAM permet d'atteindre un débit mémoire de 108 GB/s. La loi de Little [LG08] présentée dans la Annexe A.3.2.4 ne permet pas à un seul coeur de saturer la totalité du bus mémoire. Nous montrons à travers cette expérimentation qu'il faut au moins 15 coeurs pour le saturer. Les coeurs supplémentaires ne permettent pas ensuite d'améliorer le débit mémoire, car le bus est saturé. Pour des codes memory bound, il peut alors être intéressant d'en désactiver certains ou de ne pas investir dans des processeurs avec plus de coeurs. Nous présentons dans la section 3.2.3.9 un script permettant de réaliser cette recherche du nombre minimal de coeurs permettant de saturer le bus mémoire.

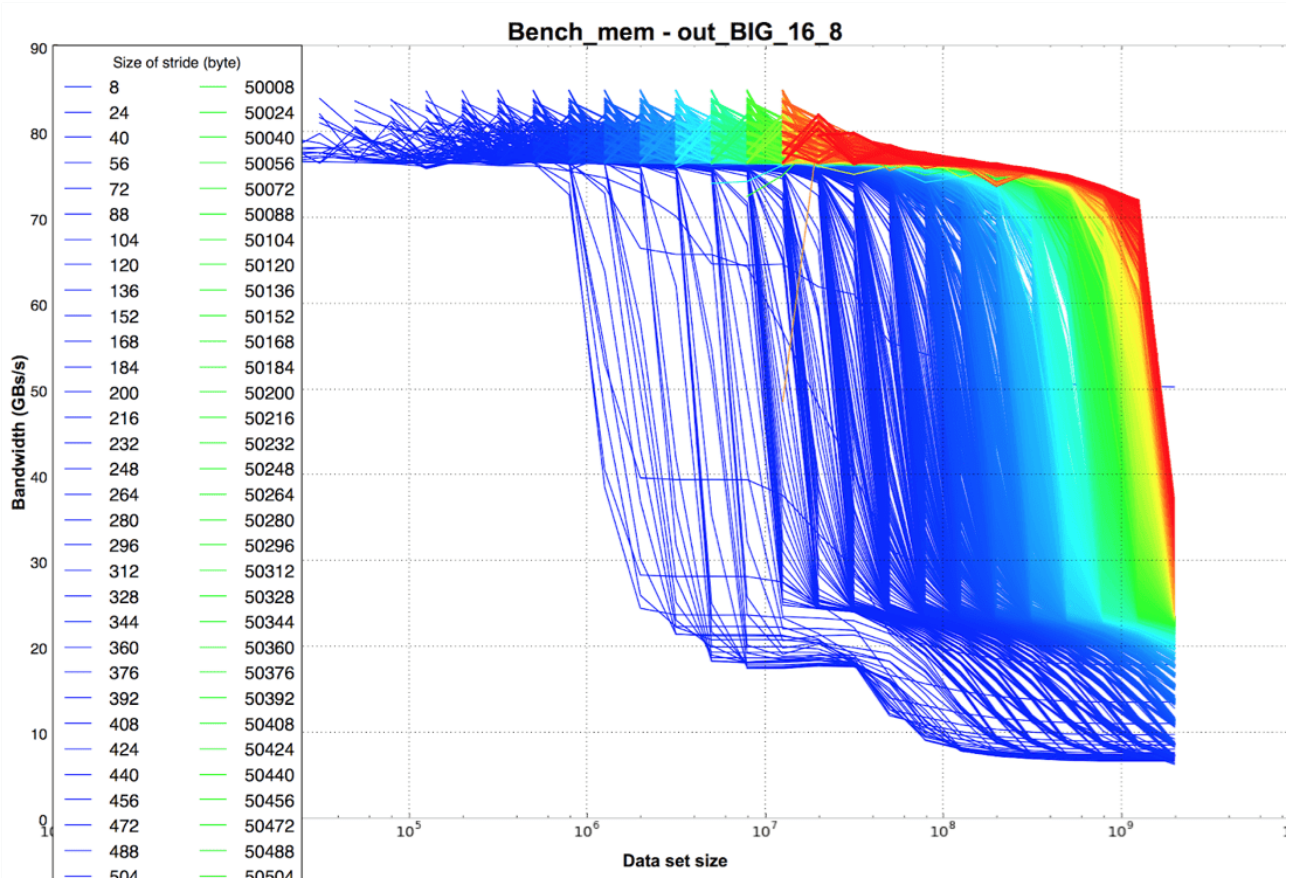


FIGURE 3.5 – Mesure du débit obtenu à l'aide du benchmark `DML_MEM`. En abscisse : taille du jeu de donnée en byte. En ordonnée : débit mesuré en GB/s. Les différentes tailles de strides sont représentées par des couleurs allant du bleu (petite taille) au rouge (grande taille). La taille des strides utilisée n'est jamais multiple de 32.

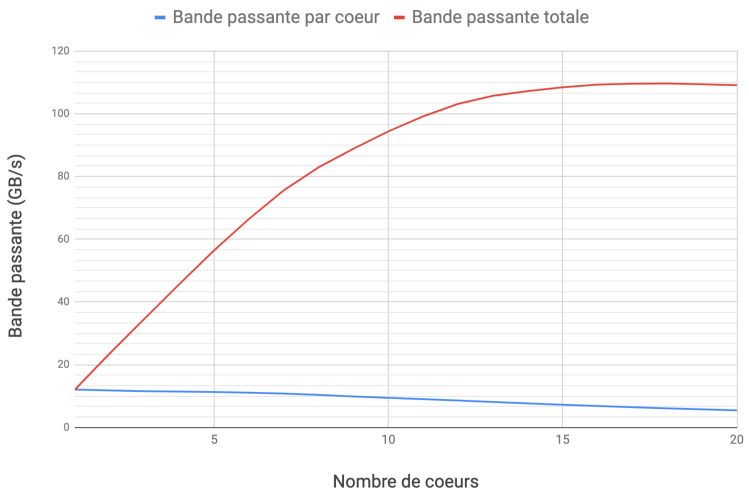
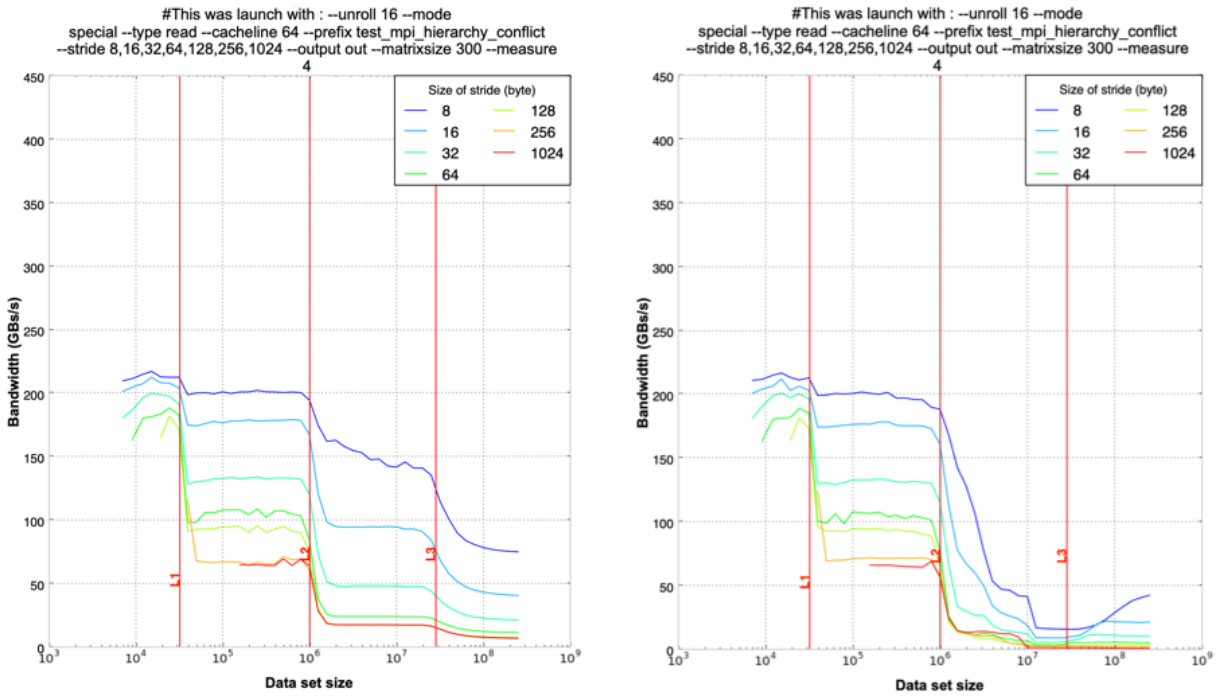


FIGURE 3.6 – Bande passante mémoire atteinte pour différents nombres de coeurs



(a) Performance mesurée lors de l'utilisation d'un seul coeur

(b) Performance mesurée lorsque les vingt coeurs du processeur sont utilisés

FIGURE 3.7 – Performance du système mémoire mesurée à l'aide du benchmark `DML_MEM` lors de sont exécution par 1 coeur (a) et vingt coeurs (b). Les résultats montrent que les caches L1 et L2 ne sont pas affectés par l'utilisation d'autres coeurs. Le débit mémoire disponible par coeur s'effondre lorsque plusieurs coeurs utilisent un jeu de données situé dans le cache L3.

3.2.3.5 Vérifier le bon fonctionnement des caches

Les caches des architectures modernes se sont complexifiées et sont devenues très efficaces pour accélérer les accès mémoire. Cependant, sur des architectures différentes de celles utilisées communément, il peut être intéressant de vérifier leur bon fonctionnement. Nous montrons dans cette expérimentation les tests pouvant être réalisés.

La première vérification est de s'assurer de l'indépendance des caches propres à chaque coeur. Dans le cas des processeurs Skylake, les deux premiers niveaux sont privés à chaque coeur. Nous avons implémenté une option pour annoter la taille de niveau de cache sur le graphique final. Les résultats de deux exécutions sur 1 et 20 coeurs sont présentés sur la figure 3.7. Comme attendu, la performance du benchmark n'est pas impactée lorsque le jeu de données utilisé est situé dans les caches L1 et L2. Les performances se dégradent lorsque le jeu de données commence à remplir le dernier niveau de cache, commun à tous les coeurs.

Une deuxième expérimentation pouvant être menée au niveau des caches est la vérification du fonctionnement du cache L3. Pour cela, nous utilisons un jeu de donnée dont la taille évolue jusqu'à remplir le cache L3. En parallèle, nous mesurons l'activité sur le bus mémoire avec l'outil `YAMB`, présenté dans la section 3.4. Les résultats de cette expérimentation sont montrés

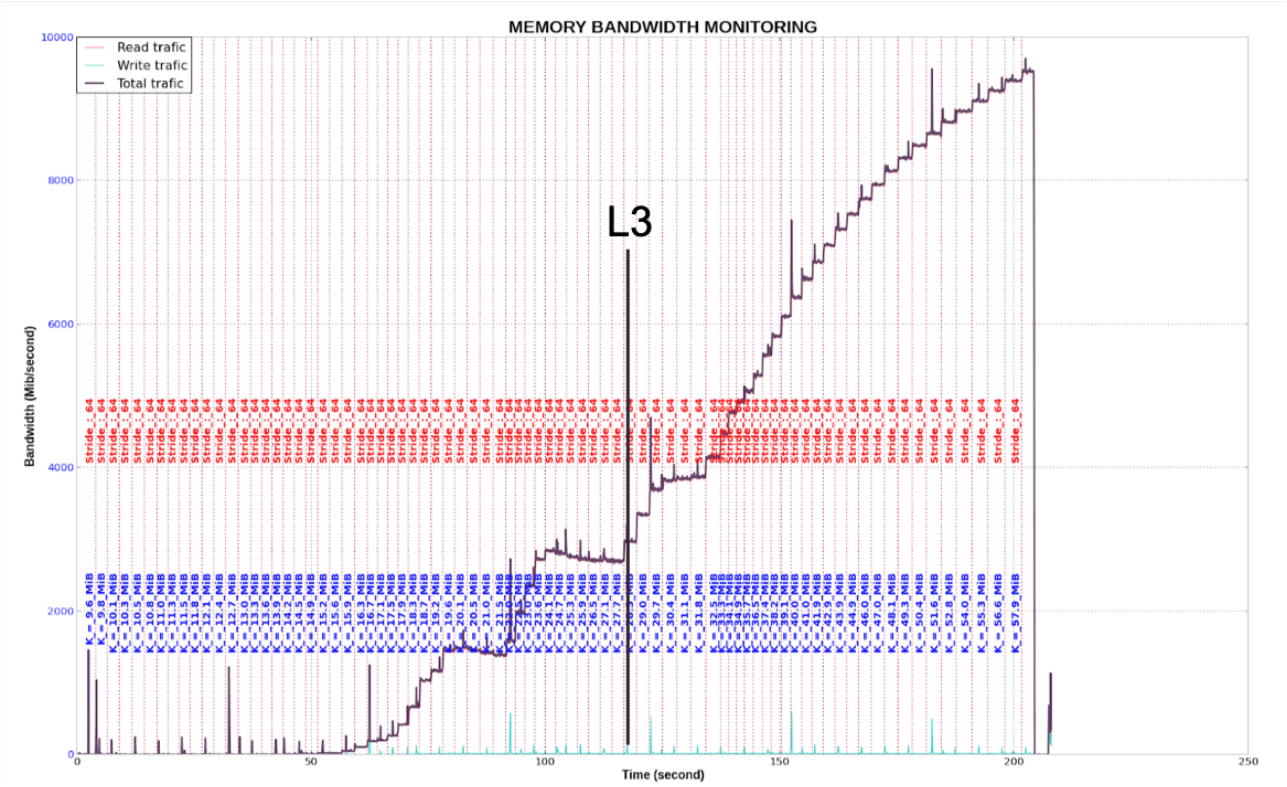


FIGURE 3.8 – Évolution de l'activité du bus mémoire en fonction de la taille du jeu de donnée.

sur la figure 3.8. Elle a été réalisée avec les commandes suivantes :

```
./monitoring_bw_main.sh --start
./dml --steplog 0.01 --unroll 2 --type read --cacheline 64 --stride 64
      --matrixsize 100 --measure 1000 --minlog 6.1 --annotate log_mem.annotate
./monitoring_bw_main.sh --stop
```

Nous montrons ainsi qu'un cache L3 de 28 MiB ne parvient pas à garder la totalité d'un jeu de donnée dépassant les 16 MiB. Au-delà de cette taille, `YAMB` mesure de l'activité sur le bus mémoire, celle-ci pouvant atteindre les 3 GB/s pour un jeu de donnée de la taille du L3 (28 MiB). Cette mauvaise utilisation du cache peut être due au phénomène de coloration de page discuté dans les expérimentations suivantes. Cette caractéristique impacte la performance de chaque coeur, car certaines données sont évincées du cache et génèrent un évènement de *miss*. Nous avons réalisé une seconde expérimentation en utilisant deux jeux de données de 20 et 28 MiB. Pour chaque jeu, nous utilisons progressivement la totalité des coeurs. Le résultat présenté sur la figure 3.9 montre que le phénomène de *trash* est encore plus fort lorsque plusieurs coeurs sont utilisés. Le trafic généré atteint les 19 GB/s pour 6 coeurs se partageant un jeu de données de 28 MiB. Cependant, la performance entre les deux benchmarks est identique, permettant de conclure au bon fonctionnement du prélecteur mémoire. Si une architecture ne possède pas un

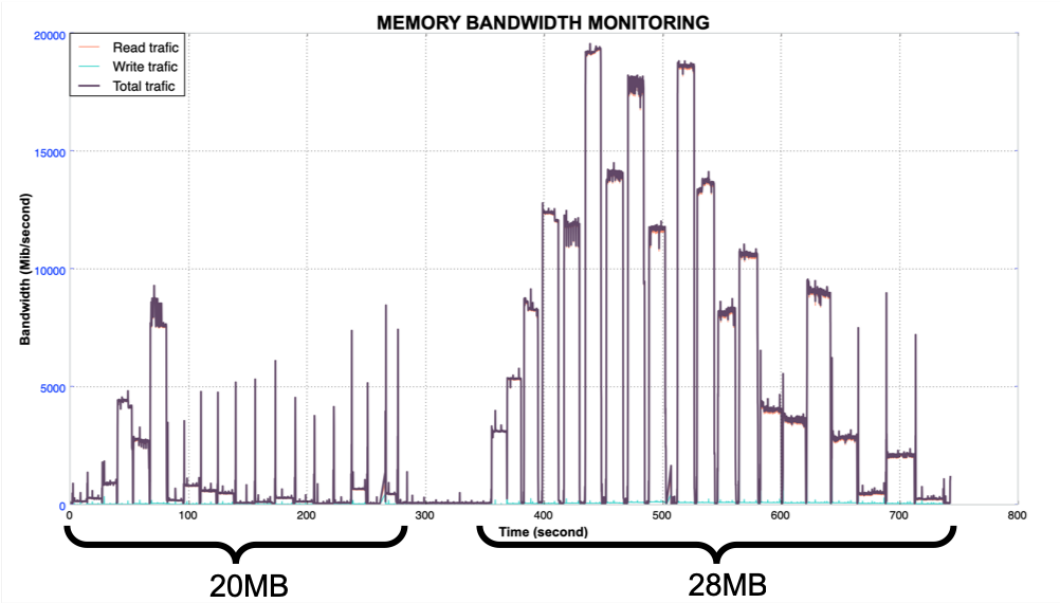


FIGURE 3.9 – Évolution du trafic mémoire pour deux jeux de données de 20 et 28 MiB. Chaque jeu est accédé par 1 à 20 coeurs.

composant aussi efficace, il peut alors être intéressant de réduire la taille des jeux de données utilisés. Pour certaines optimisations comme le *cache blocking*, nos expérimentations montrent que le plus efficace est d'utiliser 80% de la capacité du dernier niveau de cache.

3.2.3.6 Prélecteur mémoire

Dans l'expérimentation précédente, nous avons vu que le prélecteur mémoire permettait de maintenir la bonne performance du benchmark même lorsque des données sont évincées du cache. Dans cette partie, nous questionnons l'utilité de son activation permanente à l'aide de trois expérimentations :

1. débit mémoire atteignable par différents nombres de coeurs lors de l'activation ou non du prélecteur ;
2. débit mémoire atteignable par un coeur lorsque le prélecteur mémoire est activé ou non pour différentes tailles de strides ;
3. débit mémoire atteignable par différents nombre de coeurs lorsque le prélecteur est activé ou non pour une taille de *stride* correspondant à la taille de deux lignes de cache.

Impact de l'activation ou non du prélecteur. Si une architecture possède un prélecteur mémoire défaillant, il peut être intéressant de vérifier si les coeurs du processeur sont capables de générer suffisamment de requêtes mémoire pour saturer le bus. En utilisant la version parallèle de `DML_MEM`, nous avons mesuré la performance du benchmark lorsque le mécanisme de prélecture était actif ou non, avec différents nombres de coeurs. Le benchmark a été configuré

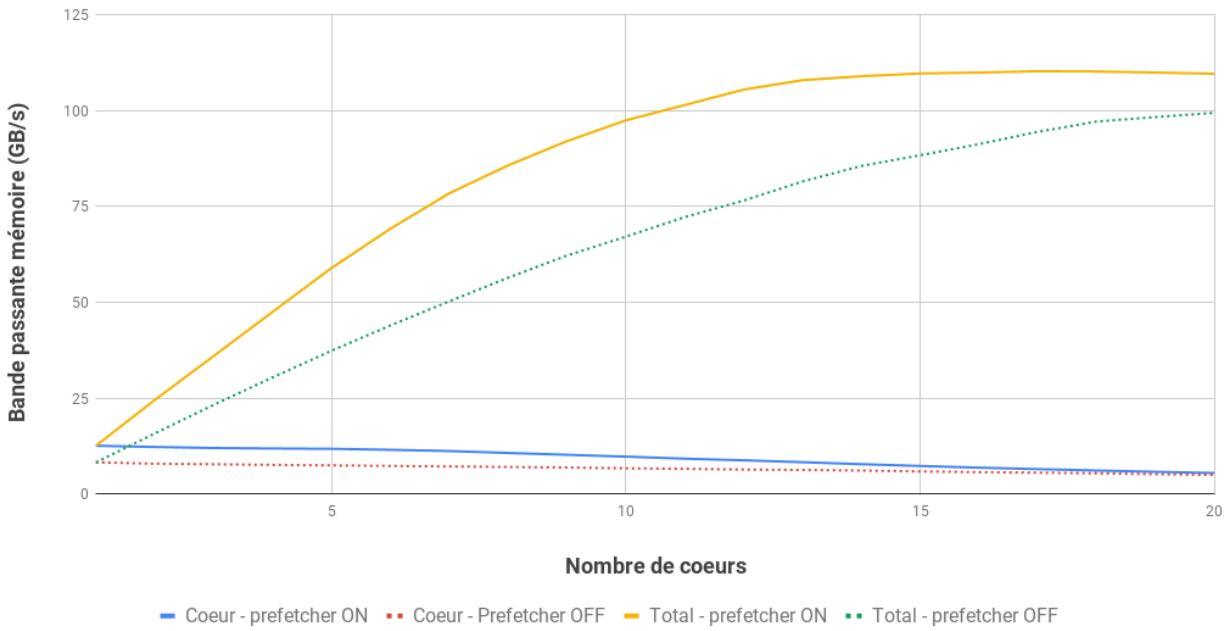


FIGURE 3.10 – Performance du benchmark pour un jeu de données de 300 MiB avec un et plusieurs coeurs actifs. Chaque mesure a été réalisée avec le prélecteur adjacent actif ou non. La taille de la stride utilisée est égale à la taille d'une ligne de cache (64 bytes).

pour utiliser des sauts de 64 bytes (une ligne de cache). Ce type d'accès correspond à un algorithme de *stream* accédant à toutes les données. La figure 3.10 montre que si le prélecteur mémoire est désactivé la totalité des coeurs ne suffit pas à saturer le bus mémoire. Dans ce cas-là, il peut alors être intéressant de réaliser le préchargement manuellement.

Utilisation d'un coeur. Si le prélecteur est efficace pour l'exécution d'un algorithme de *stream*, il est important de vérifier son bon fonctionnement pour d'autres types d'accès. Ainsi, nous avons exécuté le benchmark `DML_MEM` pour utiliser différentes tailles de *strides* lors de l'accès au jeu de données lorsque le prélecteur était actif ou non. La figure 3.11 montre le débit mémoire atteint par un seul coeur pour des sauts allant de 64 à 3700 bytes. On remarque que les performances sont similaires sauf pour une *stride* de 64 bytes, correspondant à la taille d'une ligne de cache. Pour une telle taille de *stride*, la bande passante mémoire atteinte est réduite de 40% lorsque le prélecteur mémoire est désactivé. La raison de cette baisse vient d'un mécanisme couramment utilisé dans les architectures, appelé le prélecteur adjacent (*adjacent prefetch*). Les instructions du code parcourent souvent la mémoire de façon contiguë (données d'un tableau, instruction d'un programme). Ainsi, lors d'un accès mémoire, le prélecteur adjacent anticipe les futurs accès en chargeant aussi la ligne de cache suivante. Avec l'utilisation d'une *stride* de 128 bytes, nous remarquons que le mécanisme de prélecture n'améliore plus les performances, car il

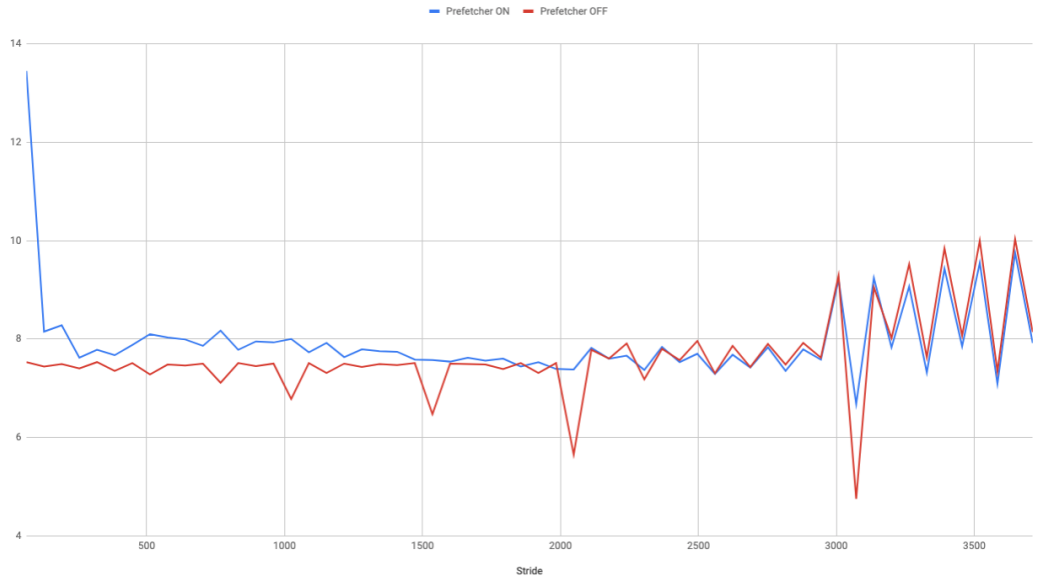


FIGURE 3.11 – Débit mémoire en GB/s (ordonnée) atteint par un coeur (en GB/s) lors de l'activation (en bleu) ou non (en rouge) du prélecteur mémoire pour différentes tailles de stride (abscisse).

ne s'occupe de charger que la ligne de cache adjacente. Lorsque celui-ci est activé, le bus mémoire transfère alors des données inutilisées pouvant affecter la performance d'une application.

Utilisation de tous les coeurs. Pour montrer que l'activation du prélecteur mémoire peut altérer les performances d'une application, la version parallèle du benchamrk `DML_MEM` a été utilisée. En effet, la prélecture de la ligne de cache adjacente n'est bénéfique que si elle est ensuite utilisée. Lorsqu'un seul coeur est actif, le chargement de cette ligne de cache n'impacte pas la performance du benchmark, car le bus est loin d'être saturé. En effet, un seul coeur n'est pas capable de générer un trafic supérieur à 10 GB/s alors que le bus mémoire peut assurer un débit dépassant 110 GB/s (voir figure 3.11).

Afin de montrer l'impact de l'activation du prélecteur mémoire, nous avons utilisé la version parallèle de `DML_MEM` pour charger la totalité des coeurs du processeur. Le benchmark a alors été utilisé pour réaliser un accès mémoire avec un saut de 128 bytes correspondant à deux lignes de cache. Ainsi, nous avons pu mesurer l'impact du prélecteur mémoire qui utilise le bus mémoire pour transférer 50% de données inutiles (stride de 128 bytes). Ainsi, les performances atteintes lorsque le prélecteur est désactivé sont augmentées de 16% (91 GB/s contre 73 GB/s). Ce type d'accès par stride plus grande que deux lignes de cache, est très courant dans les applications. Il peut donc être avantageux de désactiver le prélecteur de la ligne de cache adjacente pour ces portions de codes. De plus, le prélecteur de données peut être réalisé manuellement grâce à des instructions telles que `__builtin_prefetch (&a[i+j]);` [4].

4. Cette fonction est utilisée pour minimiser la latence en déplaçant les données dans un cache avant d'y

3.2.3.7 Déroulement de boucle

Pour améliorer les performances du benchmark `DML_MEM`, l'optimisation du déroulement de boucle a été utilisée. Au vu de l'amélioration des performances obtenues, cette section présente comment l'optimisation est implémentée et comment les applications peuvent en tirer partie. Le déroulement de boucle est une optimisation permettant de réduire l'impact du code responsable du contrôle de la boucle (le test de continuité et l'incrémentation). Le principe est de dérouler manuellement le code de plusieurs itérations dans la boucle et d'incrémenter le compteur du même nombre de déroulements. Ainsi, la proportion de code responsable du contrôle de la boucle est réduite par rapport à la proportion de code à l'intérieur de la boucle.

Première implémentation. Une première version du code utilisé pour réaliser le déroulement est présentée dans l'extrait 3.1. Pour permettre au processeur commencer les accès mémoire grâce au mécanisme d'exécution dans le désordre, 4 pointeurs différents sont utilisés. Chaque pointeur pointe vers une stride et la valeur est sommée dans la variable `sum`.

```
1  for (rep = 0; rep < repeat; rep++) {
2      DML_DATA_TYPE *p1 = mat;
3      DML_DATA_TYPE *p2 = p1 + step;
4      DML_DATA_TYPE *p3 = p2 + step;
5      DML_DATA_TYPE *p4 = p3 + step;
6      for (steps = 0; steps < ops_per_scan; steps++) {
7          sum += *p1;
8          p1 += xstep;
9          sum += *p2;
10         p2 += xstep;
11         sum += *p3;
12         p3 += xstep;
13         sum += *p4;
14         p4 += xstep;
15     }
16 }
```

Extrait 3.1 – Première version du déroulement de la boucle par 4.

Les résultats obtenus par cette première version sont mauvais, notamment pour des jeux de données situés dans les premiers niveaux de cache (baisse de la performance d'un facteur trois). Cet effondrement de performance vient de l'incapacité du compilateur à appliquer sa propre optimisation de déroulement de boucle. Par ce premier résultat, nous souhaitons attirer l'attention du programmeur sur le fait que le compilateur réalise déjà certaines optimisations. Il peut être contre-productif de les réaliser soi-même.

Deuxième implémentation. L'erreur dans la première version du déroulement est l'utilisation d'une unique variable de sommation (`sum`). En effet, cette variable doit être incrémentée à chaque lecture d'une stride et crée une dépendance en écriture, empêchant le code d'exécuter

accéder - https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

deux opérations d'addition par cycle. L'extrait 3.2 présente la version *special* de l'optimisation du déroulement qui utilise autant de variables de sommation que de déroulements réalisés. Avec cette version, la performance du benchmark dans les caches L1 et L2 est améliorée respectivement de 12% et 30% par rapport à la version non optimisée (produite par le compilateur).

```
1  for (rep = 0; rep < repeat; rep++) {
2      BM_DATA_TYPE *p1 = mat;
3      BM_DATA_TYPE *p2 = p1 + step;
4      BM_DATA_TYPE *p3 = p2 + step;
5      BM_DATA_TYPE *p4 = p3 + step;
6      for (steps = 0; steps < ops_per_scan; steps++) {
7          sum1 += *p1;
8          p1 += xstep;
9          sum2 += *p2;
10         p2 += xstep;
11         sum3 += *p3;
12         p3 += xstep;
13         sum4 += *p4;
14         p4 += xstep;
15     }
16 }
17 return sum1 + sum2 + sum3 + sum4;
```

Extrait 3.2 – Deuxième version du déroulement par 4. Cette version utilise 4 variables indépendantes pour réaliser la sommation.

Recherche du nombre optimal de déroulements de la boucle. La suite de cette expérimentation s'intéresse au nombre de déroulements de la boucle et de son impact sur la performance du benchmark. Pour cela, le benchmark a été exécuté en utilisant entre 1 et 64 déroulements sur un jeu de données remplissant 80% du cache de niveau 2. Les résultats obtenus sont présentés dans le tableau 3.4. Nous vérifions que l'optimisation du déroulement est bénéfique pour la performance du benchmark, ce qui améliore les performances progressivement de 117.8 GB/s à 337 GB/s pour un déroulement de boucle allant de 2 à 8 fois. On remarque que moins d'instructions sont exécutées et que le nombre d'instructions exécutées chaque cycle augmente de 2 à 3.22. Au-delà de 8 déroulements, la performance du benchmark se dégrade progressivement. Avec 64 déroulements de la boucle, le nombre d'instructions exécutées double et les performances s'effondrent à 140 GB/s. Cette expérimentation nous permet de montrer que la mesure de l'IPC n'est pas un indicateur suffisant pour évaluer la performance d'un code.

Pour comprendre la mauvaise performance du code déroulé plus de 8 fois, nous avons extrait leur code assembleur. L'extrait 3.3 montre comment le code du benchmark est généré avec 4 déroulements. On remarque que les 4 opérations d'addition sont vectorisées et qu'elles se suivent dans le code, permettant au mécanisme d'exécution dans le désordre de les exécuter deux par deux. On remarque aussi que 16 des 32 registres `%xmm` sont utilisés. L'extrait 3.4 expose le code assembleur du benchmark déroulant 16 fois la boucle. Le processeur utilise alors la totalité des 32 registres `%xmm`. Cependant, ce n'est pas suffisant pour réaliser tous les traitements et

Nb. de déroulements	Bande passante (GB/s)	Nb. d'instructions exécutées	Instruction par cycle
1	117.8	4204612342	2
2	235.47	3159994082	2.99
4	311.07	2634059908	3.28
8	337.96	2380891060	3.22
16	306.66	2483260088	3.07
32	315.17	2513084995	3.19
64	140.98	4993000253	2.87

Tableau 3.4 – Performance du benchmark `DML_MEM` utilisant plusieurs tailles de déroulement de la boucle pour un jeu de donnée atteignant 80% du cache L2 pour une stride de 64 byte.

il est obligé de stocker et charger certains éléments depuis la pile. L'exécution des opérations d'addition vectorisées est alors ralentie. Lorsque 64 déroulements sont utilisés, le compilateur ne parvient plus à générer d'additions vectorielles. Les performances sont alors divisées par deux, alors que l'IPC est presque similaire.

```
1  vmovhpd (%r14,%r11,8),%xmm8,%xmm9
2  lea     (%rsi,%r12,8),%r14
3  vmovhpd (%r15,%r11,8),%xmm10,%xmm11
4  lea     (%r12,%r11,2),%r12
5  vmovsd  (%r15),%xmm14
6  vmovhpd (%r14,%r11,8),%xmm12,%xmm13
7  vmovhpd (%r15,%r11,8),%xmm14,%xmm15
8  vaddpd  %xmm9,%xmm3,%xmm3
9  vaddpd  %xmm11,%xmm2,%xmm2
10 vaddpd  %xmm13,%xmm1,%xmm1
11 vaddpd  %xmm15,%xmm0,%xmm0
```

Extrait 3.3 – Boucle déroulée 4 fois.

```
1  vmovsd  (%r15),%xmm30
2  vmovhpd (%r15,%r9,8),%xmm30,%xmm30
3  lea     (%r14,%rdx,8),%r15
4  vaddpd  %xmm30,%xmm31,%xmm31
5  vmovsd  (%r15),%xmm30
6  vmovhpd (%r15,%r9,8),%xmm30,%xmm30
7  lea     (%r11,%rdx,8),%r15
8  vaddpd  %xmm30,%xmm3,%xmm3
9  .
10 .
11 .
```

Extrait 3.4 – Boucle déroulée 16 fois.

Pour mieux apprécier les performances de chaque déroulement, le benchmark est exécuté dans les différents niveaux de la hiérarchie mémoire. Les résultats sont présentés sur le graphique de la figure 3.12. Dans le cache L1, l'utilisation de 4 et 8 pointeurs permet d'améliorer la performance de 40 GB/s. On remarquera aussi la mauvaise performance de deux déroulements de la boucle dans ces premiers niveaux de caches. Cette expérimentation permet de montrer que la manière optimale d'accéder à un jeu de données présent dans les caches est d'utiliser entre 4 et 8 pointeurs différents. Au-delà, le nombre restreint de registres disponibles pour le processeur détériore la performance. La performance dans les caches des versions avec 4, 8 ou 16 déroulements est supérieure à celle du compilateur `ICC` (correspondant à un déroulement de 1 dans la figure 3.12). Lorsque les données sont dans la mémoire, le déroulement n'améliore pas les performances (voire les dégrade).

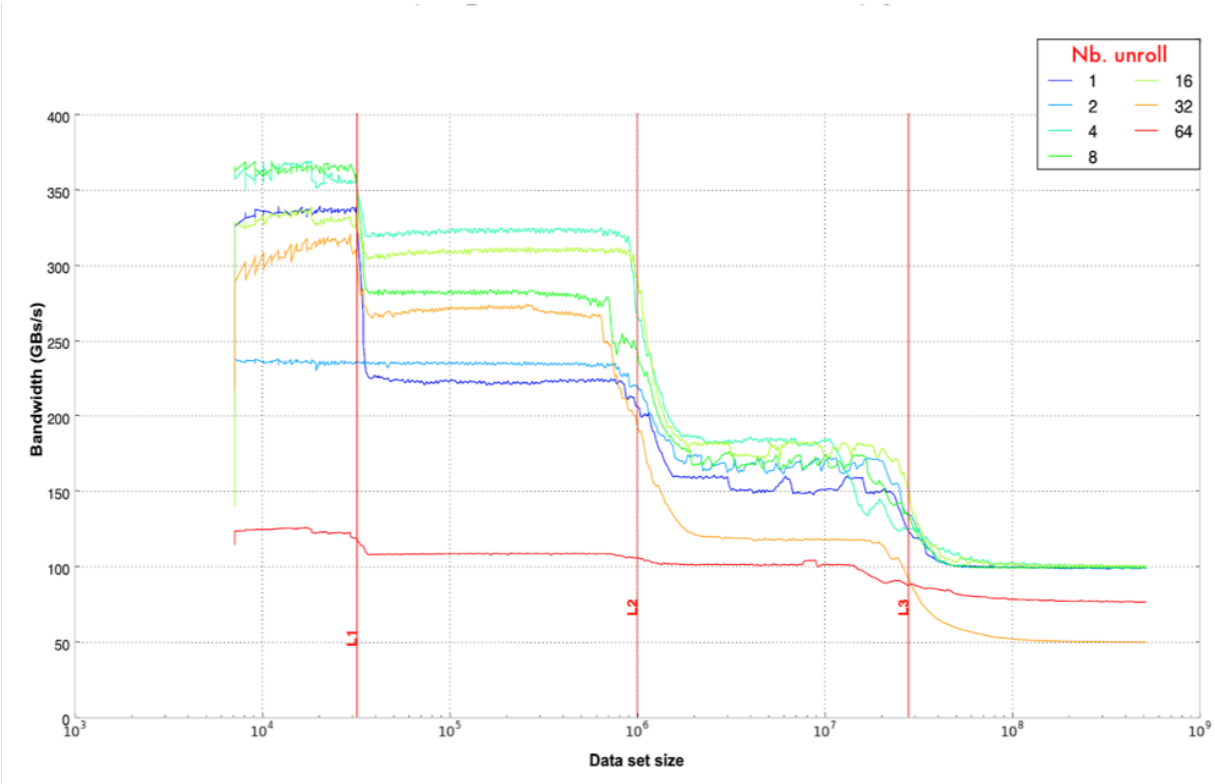


FIGURE 3.12 – Performance de plusieurs déroulements pour une stride de 8 bytes (lecture de tous les éléments).

	L1 (GB/s)	L2 (GB/s)	L3 (GB/s)	Memoire (GB/s)	dTLB-load-misses
Page de 4 KiB	320	220	150	13.40	2254025
Page de 2 MiB	340	225	200	13.45	6556

Tableau 3.5 – Performance du benchmark `DML_MEM` utilisant deux tailles de pages. Le débit des 4 niveaux de la hiérarchie mémoire (GB/s) a été mesuré en utilisant une stride de 64 bytes. Le nombre d'évènements miss correspond à l'exécution du benchmark avec un jeu de donnée situé dans le cache L3.

3.2.3.8 Large page

L'expérimentation suivante a pour objectif de comparer la performance du benchmark lors de l'utilisation de différentes tailles de pages mémoire. Comme présenté dans l'Annexe A.4.1, l'utilisation de page de plus grande taille permet d'améliorer la performance du système mémoire, notamment en accélérant le traitement de la Translation Lookaside Buffer. Les pages plus larges permettent aussi de réduire les conflits d'associativité dans les caches. L'expérimentation réalisée avec un coeur actif a permis d'obtenir les résultats présentés dans le tableau 3.5. La performance des caches est améliorée avec l'utilisation des pages larges, jusqu'à 30% dans le cache L3. Lorsque le jeu de données est dans le cache de niveau 3, la mesure du nombre d'évènements miss de la Translation Lookaside Buffer augmente d'un facteur 300. Cependant, la Translation Lookaside Buffer arrive à masquer la majorité de ces miss et conserve une bonne performance (150 GB/s).

Nous nous sommes ensuite intéressés aux performances du benchmark lorsque plusieurs coeurs sont actifs, avec et sans l'utilisation des pages larges. Nous observons les mêmes écarts de performances lorsque le jeu de données est stocké dans les caches. Lors de l'utilisation de pages standards (4 KiB), nous constatons que lorsque la taille du jeu de données approche de la taille d'un niveau de cache, la performance commence à se détériorer. Avec l'utilisation des pages de 4 MiB, la performance dans chaque niveau de cache est constante et ne se détériore que lorsque le jeu de donnée dépasse la taille du niveau de cache. Cet effet observé lors de l'utilisation de petites pages est appelé *page coloring*[5] [ZDS09].

Le graphique de la figure 3.13 montre les performances du benchmark avec un ou plusieurs coeurs actifs sur un jeu de donnée de 2 GiB. Nous remarquons que la performance du benchmark baisse lorsque plus de 15 coeurs sont utilisés avec des pages larges. La performance par coeur diminue de 5.39 GB/s à 3.44 GB/s alors que le bus mémoire n'est pas saturé. Nous n'avons pour le moment pas réussi à expliquer ce problème qui affecte tous les jeux de données supérieurs à 2 GiB.

Bien que la Translation Lookaside Buffer limite rarement les performances des applications réelles, le recours à l'utilisation de pages larges peut être bénéfique. Les versions récentes du

5. La coloration de page est une technique logicielle permettant d'améliorer le mappage d'une page de la mémoire physique sur les lignes de cache du processeur. Cette technique permet d'assurer que des pages contiguës en mémoire virtuelle sont allouées à des pages physiques qui se répartiront efficacement dans les caches.

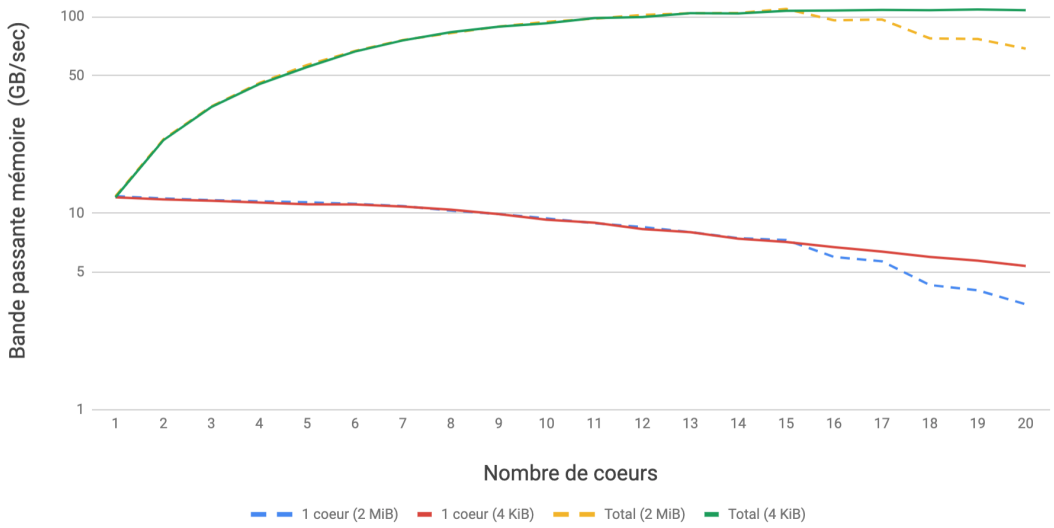


FIGURE 3.13 – Débit mémoire mesuré en GB/s atteint par un coeur (rouge et bleu) et par différents nombres de coeurs (jaune et vert) utilisant des pages de 4 KiB (rouge et vert) et 2 MiB (bleu et jaune) pour accéder à un jeu de données de 2 GiB.

noyau Linux peuvent choisir elles-mêmes lors de l'allocation mémoire si le recours à des pages plus grandes peut être bénéfique pour l'application (Red Hat Transparent Huge Pages (THP)). Comme nous montrons dans cette dernière expérience que l'utilisation de pages de 2 MiB peut détériorer les performances, nous préconisons que ce choix revienne à l'utilisateur. Il devra décider ou non de leur utilisation, en fonction de l'application utilisée.

3.2.3.9 Fréquence et coeurs : impact sur le débit mémoire

Dans cette expérimentation nous avons mesuré la bande passante mémoire atteignable par notre benchmark pour différente configuration de fréquence et de nombres de coeurs actifs. Les résultats sont visibles sur la figure 3.14. Comme dans l'expérimentation précédente (section 3.2.3.4), nous démontrons que la totalité des coeurs n'est pas nécessaire pour saturer la bande passante. Cette expérience montre aussi que les coeurs utilisés n'ont pas besoin d'utiliser leur fréquence maximale. En effet, notre benchmark sature le bus mémoire avec 17 coeurs cadencés à seulement 2.1 GHz. Le script utilisé pour générer la figure 3.14 annote automatiquement le graphique pour identifier rapidement les maximums. Une telle utilisation du benchmark `DML_MEM` permet d'identifier le couple `{fréquence, nombre de coeurs}` minimal pour saturer le bus mémoire. Pour des applications limitées par la performance de ce dernier, il peut être intéressant de désactiver les coeurs supplémentaires ou de plafonner la fréquence du processeur pour limiter la consommation électrique. En plus de vérifier la présence potentielle de bogues dans l'architecture, cette expérimentation peut permettre à un utilisateur de choisir la meilleure configuration pour un achat de nouveaux processeurs.

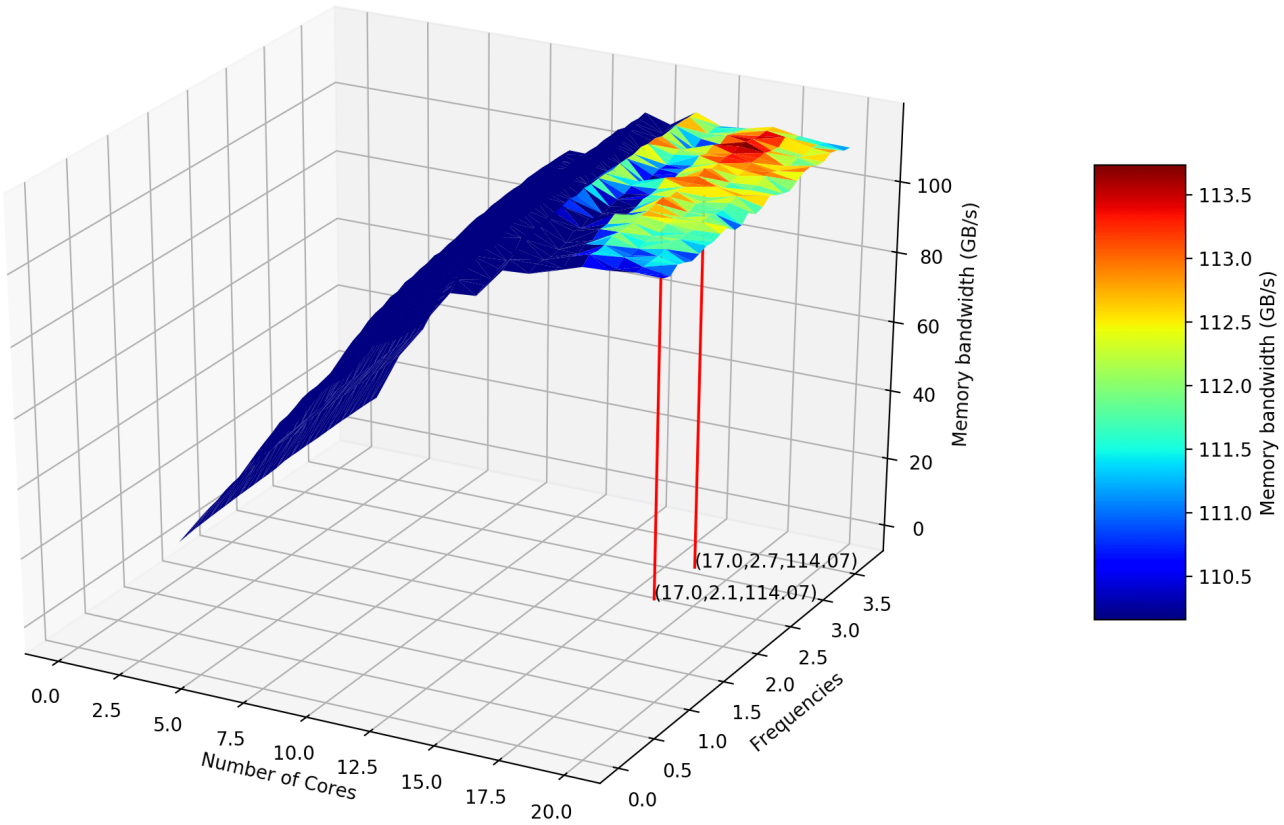


FIGURE 3.14 – Mesure du débit du bus mémoire (en GB/s) atteint par le benchmark lors de l'utilisation de différents nombres de coeurs dont la fréquence est comprise entre 1.5 GHz et 3.5 GHz. Le débit maximal atteint pour ce processeur est de 114.07 GB/s en utilisant 17 coeurs cadencés à 2,1 ou 2,7 GHz.

3.2.4 Conclusion

Cette section s'intéresse au benchmark `DML_MEM`, un outil permettant de caractériser de multiples parties du système mémoire. Le benchmark utilise des accès mémoire par saut, appelés strides et nous a permis de caractériser plusieurs parties de la microarchitecture. Une première utilisation nous a permis de déduire la taille d'une ligne de cache, donnée importante pour la programmation efficace d'applications. En utilisant différentes tailles de strides, nous avons pu mesurer l'impact que celles-ci pouvaient avoir sur la performance des différents niveaux de la hiérarchie mémoire. En utilisant une taille de stride correspondant à la taille d'une ligne de cache, nous avons pu caractériser l'architecture pour l'exécution d'algorithmes de streaming. Nous avons ainsi pu mesurer le débit maximal atteignable par le bus mémoire. Lors de nos expérimentations, nous avons obtenu des débits supérieurs (de l'ordre de 5%) à ceux obtenus grâce au benchmark `STREAM` [McC95]. Nous avons ensuite utilisé le benchmark `DML_MEM` pour exposer la complexité des microarchitectures modernes. Nous montrons que la microarchitecture peut avoir des comportements inattendus lors de l'utilisation de certaines tailles de saut. À travers

ces expérimentations, nous souhaitons attirer l'attention du programmeur sur la complexité de la microarchitecture et de la nécessité de sa caractérisation. Nous avons pu voir que les performances atteintes par un code aussi simple pouvaient varier de plusieurs ordres de grandeur à cause de paramètres subtils : préchargement mémoire, taille des pages, nombre de déroulements de boucle...

Lors des travaux de thèse, l'utilisation de cet outil nous a permis de caractériser plusieurs plateformes différentes et de déceler un bogue majeur dans un nouvel accélérateur. Ce bogue, inconnu par l'entreprise l'ayant conçue, condamne les applications utilisant des motifs de calculs de type Stencil, à n'accéder qu'à une fraction des performances mémoire théoriques de l'accélérateur. Ce travail réalisé pour un client industriel ne peut cependant pas être rendu public dans ce manuscrit.

3.3 Benchmark d'unité arithmétique

Cette section présente le développement de notre benchmark appelé `Kernel Generator`. Cet outil permet de vérifier le bon comportement du matériel responsable de l'exécution des instructions de calculs sur des nombres à virgule flottante, utilisées par la grande majorité des applications HPC. L'exécution de ces opérations est réalisée par un composant, appelé unité de calcul en virgule flottante (ou unité de calcul à virgule flottante (FPU)), présentée dans la section suivante. Lorsque la performance des applications n'est pas limitée par celle du système mémoire, leur performance dépend essentiellement de celle des FPU.

Cet outil a fait l'objet d'une publication à la conférence HPCS 2019 :
Jean Pourroy*, Patrick* Demichel *et Denis* Christophe*. « Assembly micro-benchmark generator for characterizing Floating Point Units ». In :* HPCS 2019 - 17th International Conference on High Performance Computing & Simulation*. Dublin, Ireland : IEEE, juil. 2019*

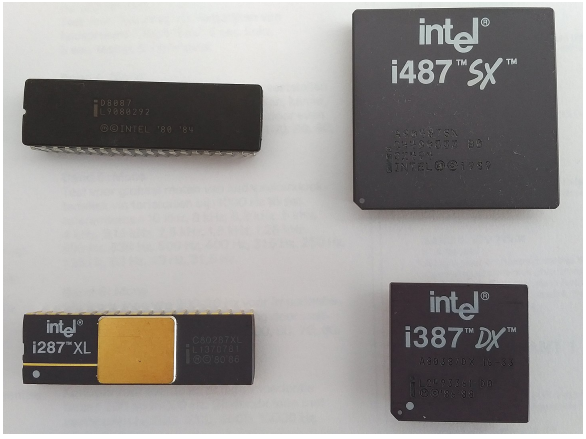
3.3.1 Motivation et objectifs

La performance des applications sur des architectures modernes est principalement limitée par celle du système mémoire. Ainsi, les efforts de développement d'outils de caractérisation ont en grande partie été consacrés à la caractérisation de cette partie de l'architecture. Cependant, avec le développement de nouvelles architectures, la différence de performance entre les unités de calculs et celle du système mémoire devrait être plus équilibrée. Il est donc important d'avoir les outils nécessaires pour les caractériser.

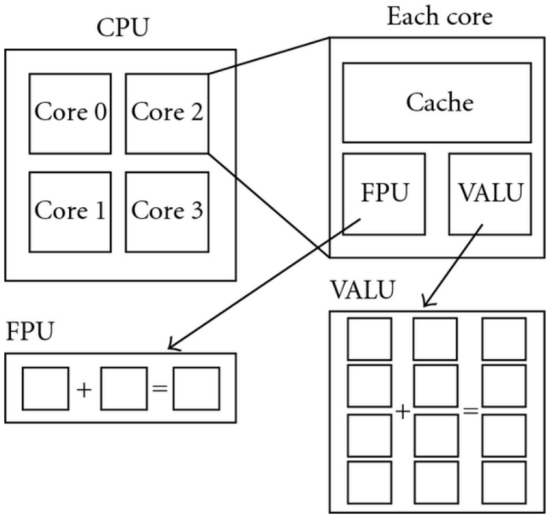
Bien que le comportement des FPU d'architectures actuelles soit connu, certaines particularités sont encore difficiles à caractériser : exécution dans le désordre et dépendances entre instructions (Annexe A.2.3.5) ou la fréquence atteignable pour un type d'instruction vectorielle (Annexe A.2.3.1). Il est nécessaire de connaître la performance maximale d'un processeur (mesurée par le nombre d'opérations à virgule flottante (FLOP)) pour différents types de calculs. Celle-ci peut alors être utilisée pour apprécier les performances d'un code grâce à des modèles de type Roofline (voir section 2.4.2.5). Afin d'obtenir le débit maximal de calculs réalisables par une architecture, deux approches peuvent être envisagées. La première utilise les caractéristiques matérielles pour le calculer. Cependant, la complexité des architectures nous a souvent montré que la performance réellement atteignable pouvait inférieure, mais parfois supérieure aux performances théoriques. La deuxième technique utilise des codes de benchmarks. En utilisant ces codes, il est alors possible de trouver des comportements cachés de l'architecture et d'en déceler des limitations.

3.3.1.1 Floating Point Unit (FPU)

La FPU est un composant majeur des ordinateurs et a connu de nombreuses évolutions au fil des générations. À l'origine, les FPU étaient des composants additionnels pouvant être ajoutés sur la carte mère pour accélérer l'exécution de calcul en virgule flottante (figure 3.15a). C'est pour cela qu'il est commun de designer ce composant comme un accélérateur. Cependant, les



(a) À l'origine, les FPU pouvaient être achetées séparément des processeurs.



(b) Aujourd'hui, la FPU fait partie de chaque coeur du processeur.

FIGURE 3.15 – La FPU d'un processeur est un matériel permettant d'exécuter efficacement des instructions de calculs arithmétiques sur des nombres à virgule flottante : addition, soustraction, multiplication, division, racine carrée, etc... [6]

applications réalisant de plus en plus de calculs de ce type, les FPU ont ensuite été directement intégrées au processeur (figure 3.15b). La FPU reçoit ses instructions du même décodeur que celui de l'unité arithmétique et logique (ALU). Lorsque les premiers étages du pipeline ont décodé l'instruction à exécuter (voir section A.2.4.1), le séquenceur choisit si elle doit être exécutée par l'ALU ou la FPU.

L'ajout de la FPU directement sur les coeurs a notamment permis de réduire les latences d'accès. De plus, avec l'apparition de plateformes multiprocesseurs, la gestion d'erreurs et d'exceptions liées aux FPU externes était devenue très difficile. Le premier processeur Intel à posséder une FPU et une ALU sur la même puce est le processeur Intel 80486DX produit en 1989. Aujourd'hui, les FPU sont des composants de haute performance capable d'exécuter plusieurs instructions de calcul sur des nombres réels grâce à des instructions vectorielles.

Les applications de HPC réalisent majoritairement des calculs sur nombres à virgule flottante qui sont exécutés par les FPU. Ces unités sont très sollicitées et il est important d'en connaître les caractéristiques : débit, latence, fréquence maximale soutenable par le processeur. Les instructions vectorielles les plus longues utilisent une plus grande quantité de transistors pour être exécutées. Ainsi, la chaleur émise par le processeur varie et les instructions les plus complexes ne peuvent pas être exécutées aux fréquences maximales du processeur. Les différentes fréquences atteignables pour un type d'instruction peuvent être difficiles à prévoir.

6. Source images : https://en.wikipedia.org/wiki/Floating-point_unit et https://www.extremetech.com/computing/263963-intel-reverses-declares-skylake-x-cpus-two-avx-512-units

3.3.1.2 Objectifs

Dans la suite de cette section, nous présentons un outil permettant de caractériser finement les performances des unités de calcul en virgule flottante (FPU). Le développement de cet outil a été motivé par la nécessité d'obtenir différentes caractéristiques de la microarchitecture :

1. La première caractéristique pouvant être mesurée est la performance maximale atteignable par la FPU pour un certain type d'instruction. Cette performance est mesurée en nombre opérations à virgule flottante par seconde (FLOPS) et notée $\texttt{FLOPS}_{\texttt{max}}$). Elle doit être mesurée pour des types et des tailles d'instructions vectorielles différents. L'avantage de cette unité est de rendre les résultats du `Kernel Generator` comparable avec ceux d'autres benchmarks (tel que `HPL` [DLP03]).
2. La deuxième caractéristique mesurable est la latence des instructions. Il peut être intéressant de connaître celle-ci lors de l'exécution d'instructions dépendantes.
3. D'autres comportements doivent être caractérisés pour anticiper et comprendre la performance de certaines applications. Parmi eux, celui de l'unité d'exécution dans le désordre. Certains codes comportant beaucoup de dépendances sont limités par la faculté du processeur à exécuter plusieurs chaînes de dépendances.

3.3.2 Kernel Generator

Afin de répondre aux objectifs fixés dans la section précédente, nous avons développé l'outil `Kernel Generator`. Il permet de générer des kernel en assembleur pour caractériser finement le comportement des FPU. Cet outil vise plus particulièrement les développeurs d'application dont la performance est limitée par le débit d'exécution d'instructions de calcul (compute bound). Bien que la performance des processeurs soit souvent limitée par la bande passante mémoire, une partie significative des codes est purement compute bound.

L'outil du `Kernel Generator` permet de mesurer certaines caractéristiques de la FPU d'un processeur grâce à l'exécution d'un kernel généré en assembleur comportant des instructions de calcul arithmétique. Les instructions utilisées peuvent être scalaires ou vectorielles. La version actuelle de l'outil supporte les instructions de type scalaire, SSE, AVX2 et AVX512.

Grâce à ses différentes options, l'utilisateur peut générer des kernels d'instructions de type et de tailles différents. La valeur de l'outil vient de son utilisation et des différents tests que le programmeur peut réaliser. L'outil peut aussi être utilisé pour détecter des problèmes de la microarchitecture lors de l'exécution intensive d'instructions de calculs. Détecter ces comportements cachés peut ensuite permettre de mieux apprécier la performance d'une application réelle.

3.3.2.1 Concept

L'outil `Kernel Generator` est un générateur de code assembleur permettant de mesurer la performance de la FPU d'un coeur de processeur. Pour cela, l'outil génère automatiquement un programme en langage C++ contenant :

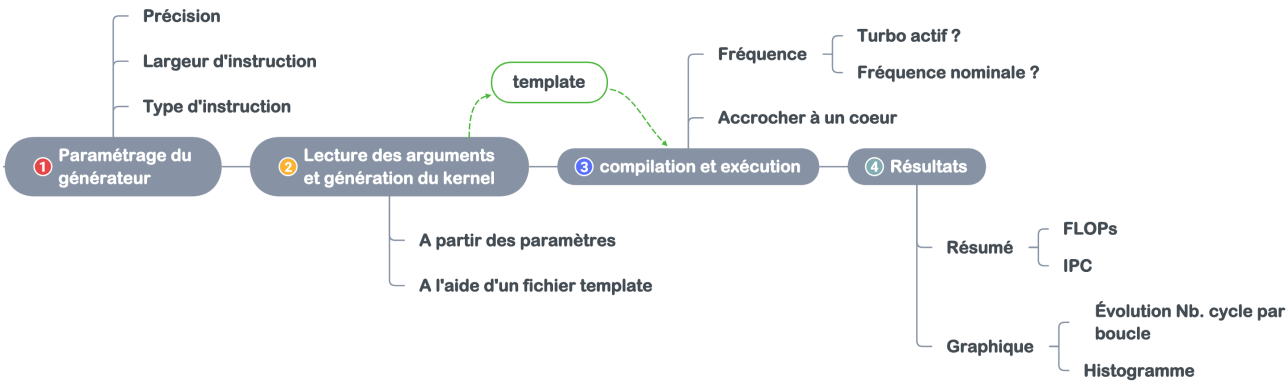


FIGURE 3.16 – Déroulement de l'exécution du `Kernel Generator` : le générateurs lit les paramètres et génère à l'aide d'un fichier template le benchmark. Une fois compilé, le programme généré est exécuté et les résultats sont affichés à l'utilisateur.

— Les instructions assembleurs sélectionnées par l'utilisateur.
— Le code responsable de l'exécution et des différentes mesures de performance

La figure 3.16 montre les étapes principales de l'utilisation du `Kernel Generator`. Le code généré par l'outil reste simple afin de faciliter l'établissement de conclusions par utilisateur. En effet, produire un code trop complexe rendrait l'analyse des résultats trop difficile sur des architectures aussi complexes que celles étudiées. Un exemple d'utilisation de l'outil peut être réalisé grâce à la commande suivante :

```
./kg   -P double -W 128  -O aamm
```

Cette commande permet de générer un kernel de calculs utilisant quatre instructions vectorielles de 128 bits sur des nombres flottants en double précision. Le kernel généré est présenté dans l'extrait 3.5. Il est composé de quatre instructions : deux additions et deux multiplications.

```
1 "vaddpd %%xmm0, %%xmm1, %%xmm2;"
2 "vaddpd %%xmm0, %%xmm1, %%xmm3;"
3 "vmulpd %%xmm0, %%xmm1, %%xmm4;"
4 "vmulpd %%xmm0, %%xmm1, %%xmm5;"
```

Extrait 3.5 – Exemple de kernel généré à l'aide de la commande `./kg -P double -W 128 -O aamm` contenant 4 instructions vectorielles de 128 bits.

À partir des arguments, le générateur écrit un programme en langage C++ qu'il compile et exécute. Les résultats sont ensuite présentés sous forme de texte dans le terminal ou sous forme de graphique (nécessite Python). Le choix du langage assembleur a été motivé par la volonté de s'assurer que les instructions générées étaient bien exécutées. En effet, les compilateurs deviennent toujours plus performants et optimisent les codes rendant l'analyse de performance plus difficile. De plus, ils peuvent détecter l'artificialité d'un code et contourner son exécution. En générant nous même le code assembleur, nous nous assurons que la performance du code mesuré est bien celle du code attendu.

3.3.2.2 Les options.

Une des forces du `Kernel Generator` vient de sa capacité à générer un grand nombre de benchmarks différents. Ceci est possible grâce à l'utilisation des différentes options dont les principales sont listées ci-dessous :

`--operation [a,m,f]+` Lister les opérations à réaliser par le kernel à l'aide d'une chaîne de caractères composée des lettres `a`, `m` et `f` correspondant aux opérations suivantes : addition (a), multiplication (m) ou une multiplication et une addition fusionnées (FMA) (f). Différentes opérations peuvent être mixées et seront insérées dans le kernel dans le même ordre. Un script externe peut alors être utilisé pour générer des kernels testant différentes combinaisons d'opérations. En faisant varier à la fois les opérations et la taille des instructions, l'outil permet de découvrir des comportements cachés des architectures.

`--precision (single | double)` Définir la précision utilisée pour réaliser les calculs : simple ou double. Pour le moment, des instructions ayant des précisions différentes ne peuvent pas être mélangées dans un même kernel.

`--width (64 | 128 | 256 | 512)` Définir la *largeur* des instructions vectorielles utilisées. Sur une architecture Intel, ces instructions correspondent aux ISA suivantes : MMX (64), SSE (128), AVX (256) et AVX-512 (512). Pour le moment, des instructions vectorielles de tailles différentes ne peuvent pas être mixées dans un même kernel.

`--dependency N` Générer une instruction dont un opérande est le résultat produit par l'instruction précédente. Un nombre `N` peut être donné pour générer plusieurs chaînes de dépendances. Cette option est particulièrement utile pour mesurer la latence des instructions et la performance du tampon d'exécution dans le désordre.

`--binding N` Accrocher (*bind*) le benchmark généré à un coeur spécifique. Le benchmark n'étant pas parallélisé, il est nécessaire d'en exécuter plusieurs versions en parallèle pour tester la performance d'un processeur lorsque plusieurs coeurs sont utilisés. Les différents processus créés peuvent alors être *accrochés* aux différents coeurs du processeur à l'aide d'un script externe.

`--unroll N` Appliquer l'optimisation du déroulement de boucle `N` fois. Cette optimisation permet de dérouler plusieurs fois le corps de la boucle à l'intérieur de celle-ci pour réduire l'impact du traitement des instructions de contrôle de la boucle (incrémentation et comparaison) sur les performances du benchmark.

`--frequency (true | false)` Générer puis exécuter une fonction en début de benchmark qui mesure la fréquence du processeur. Les calculs des résultats sont impactés par l'utilisation du mode turbo. En connaissant la valeur de la fréquence turbo, ces résultats peuvent être

ajustés. En effet, la mesure de la performance du benchmark est réalisée en utilisant la valeur de la fréquence de base du processeur (ou fréquence nominale).

`--loopsize N` Améliorer la précision des résultats et éliminer les potentiels bruits de mesure en réalisant `N` mesures.

3.3.2.3 Fonctionnement du générateur

Une fois les différentes options analysées, le générateur va écrire un programme C++ contenant le benchmark à exécuter. Tous les benchmarks ont une partie commune de code qui est stockée dans un fichier *template*. Le générateur s'occupe seulement d'écrire la partie en assembleur dans ce fichier et d'initialiser quelques variables (nombre de mesures, coeur sur lequel s'exécuter). Le *template* contient le code permettant de mesurer les résultats, de calculer la fréquence et d'afficher le résultat dans le terminal. Un exemple d'exécution du générateur est résumé sur la figure 3.17.

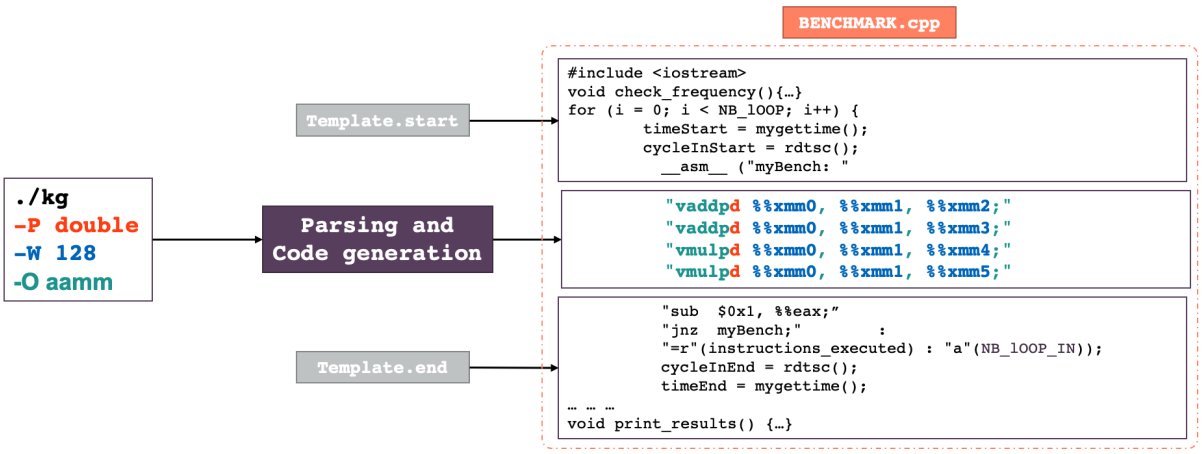


FIGURE 3.17 – Génération du benchmark à partir de la ligne de commande entrée par l'utilisateur. La partie commune du code est stockée dans un fichier de *template*.

Lorsque le code du benchmark généré, celui-ci est compilé automatiquement par le générateur (le compilateur utilisé peut facilement être modifié). Le fichier source et l'exécutable sont créés dans le dossier courant de l'utilisateur. L'exécutable peut ensuite être utilisé sans passer de nouveau par le générateur. Un exemple de résultat de l'exécution du benchmark est présenté dans l'extrait 3.6.

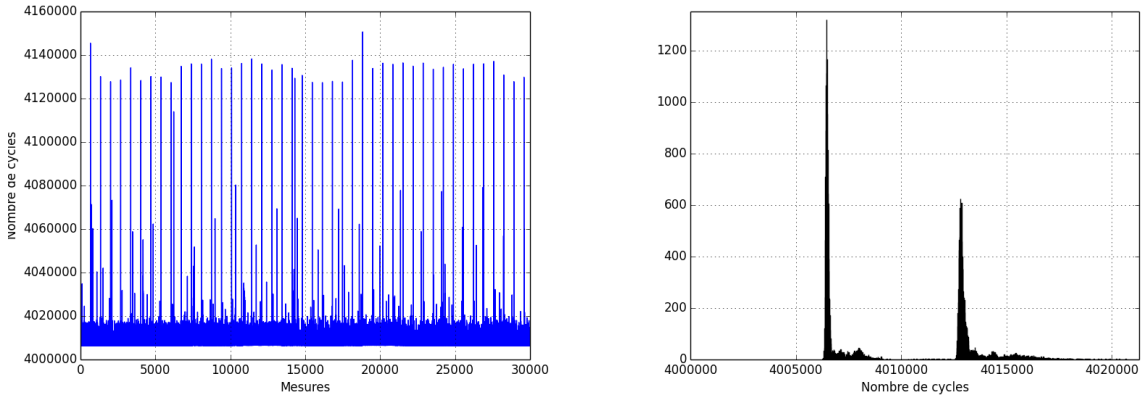
```
./kg -W 512 -O aamm -P double -U 4 -S 30000 -L 500000

------------------------- CHECK FREQUENCY -------------------------
+ Base      frequency is 2.69GHz
+ Current   frequency is 2.68GHz
+ OK: the core is running at his frequency based value

------------------------ INSTRUCTIONS SUMMARY -------------------------
_label_|   NB INSTRUCTIONS      Time    FREQUENCY   Giga_inst/sec     IPC
_value_|      240000000000      44.7         2.69            5.37    1.99

------------------------- FLOP SUMMARY -------------------------
PRECISION     FLOP/cycle          FLOP/second
   Single              0                    0
   Double             16              4.3e+10
------------------------------------------------------------------------
```

Extrait 3.6 – Exemple d'exécution d'un benchmark de quatre instructions AVX-512.

Le benchmark généré comporte quatre instructions vectorielles AVX-512 : deux additions et deux multiplications. Le processeur utilisé est un Intel Xeon 6150 cadencé à 2.70GHz. Pour cette expérimentation, la fréquence turbo a été désactivée. Pour améliorer la précision des résultats, la boucle générée est déroulée 4 fois. La boucle réalise 500 000 itérations et sa performance est mesurée 30 000 fois. Le CPU est capable d'exécuter deux opérations flottantes vectorielles par cycle d'horloge. Pour des données en doubles précisions, cela correspond à réaliser 16 opérations flottantes par cycle. La puissance de calcul atteinte par l'exécution du benchmark sur un coeur est de 43 GFLOPS. Pour chaque mesure le nombre de cycles et le temps nécessaire à l'exécution de la boucle sont sauvegardés dans un fichier. Ce fichier peut ensuite être affiché avec un script python (voir figure 3.18). La figure 3.18a permet de voir comment la performance du benchmark évolue au fil des exécutions. On peut détecter des problèmes de détérioration des performances pouvant être dus à un mauvais refroidissement du processeur par exemple. La figure 3.18b affiche l'histogramme permettant de voir deux familles de performance. Dans cet exemple, la différence entre ces deux familles est inférieure à 0,02%. Il peut cependant arriver que pour certaines architectures, le processeur ne soit pas capable de maintenir une performance constante lors de l'exécution de certaines instructions. Ce comportement pourra alors être mis en évidence grâce à ce graphique (figure 3.18b).

3.3.2.4 Validation des résultats

Lors du développement du générateur, il a été nécessaire d'utiliser des outils nous permettant de valider les performances rapportées par le benchmark. Pour s'assurer du bon fonctionnement du benchmark sur de nouvelles architectures, certaines de ces méthodes de vérification ont été implémentées dans l'outil directement.



(a) Évolution du nombre de cycles nécessaires pour l'exécution de la boucle de benchmark

(b) Histogramme du nombre de cycles nécessaires

FIGURE 3.18 – Grâce à un script python, le fichier de résultat peut être affiché sous forme de graphiques.

Validation du nombre d'instructions. La première valeur à valider est de s'assurer que le bon nombre d'instructions a été exécuté par le benchmark. Le benchmark affiche le nombre d'instructions qui devrait être exécuté. Un moyen de le vérifier est d'utiliser les compteurs matériels. L'outil `perf` permet de mesurer le nombre d'évènements d'un compteur en utilisant son adresse. La ligne de commande suivante permet de compter les opérations flottantes réalisées par le benchmark : `perf stat -e /rffc7 ./benchmark`. Le résultat du benchmark ci-dessus prévoyait l'exécution de 240,000,000,000 opérations. La commande `perf stat` retourne une valeur de 240,000,210,024 opérations (une instruction FMA compte pour deux). Pour comprendre d'où proviennent les instructions supplémentaires, nous avons mesuré différentes versions du benchmark avec différentes longueurs de boucle (5000, 10000 et 20000). Le tableau 3.6 donne les résultats affichés par le benchmark et par la commande précédente utilisant l'outil `perf`. Les résultats de `perf` mesurent le bon nombre d'instructions avec 2124 opérations supplémentaires à chaque fois. Ces instructions supplémentaires correspondent au traitement des résultats.

Commande utilisée	Nombre d'instructions attendu	Résultat perf
./kg -W 512 -O aamm -S 300 -L 5000	6000000	6002124
./kg -W 512 -O aamm -S 300 -L 10000	12000000	12002124
./kg -W 512 -O aamm -S 300 -L 20000	24000000	24002124

Tableau 3.6 – Vérification du nombre d'instructions exécutées avec l'outil `perf`.

Validation de l'IPC. La validation du nombre d'IPC du processeur peut elle aussi être réalisée grâce à `perf` : `perf stat ./benchmark`. Pour générer le benchmark la commande suivante a été utilisée :
`./kg -W 512 -O aamm -P double -S 1000 -L 90000000`
Lors de son exécution, le benchmark donne un IPC de 2 alors que la commande `perf` donne un résultat de 3. Cette différence peut être expliquée en regardant le code assembleur généré (voir extrait 3.7).

```
1  cycleInStart = rdtsc();
2  __asm__ (""
3      "myBench: "
4      "vaddpd %%zmm0, %%zmm1, %%zmm2; "
5      "vaddpd %%zmm0, %%zmm1, %%zmm3; "
6      "vmulpd %%zmm0, %%zmm1, %%zmm4; "
7      "vmulpd %%zmm0, %%zmm1, %%zmm5; "
8      "sub  $0x1, %%eax;"
9      "jnz  myBench;"   : "=r" (instructions_executed) : "a" (NB_lOOP_IN));
10 cycleInEnd = rdtsc();
```

Extrait 3.7 – Code généré par la commande ./kg -W 512 -O aamm -P double. Le benchmark mesure la performance de la boucle à l'aide du compteur de cycle (`rdtsc()`). Cette mesure comprend l'exécution des instructions de calculs du kernel mais aussi les deux instructions de la gestion de boucle `sub` et `jnz`.

Le résultat donné par le benchmark ne prend en considération que les quatre instructions de calculs pour calculer l'IPC alors que l'outil `perf` compte la totalité des instructions exécutées. Notre calcul prend pour postulat que les deux instructions de gestion de boucle (la décrémentation `sub`, et le saut conditionnel `jnz`) ne rentre pas dans le profil de l'exécution. En effet, grâce au prédicateur de branchement, l'instruction de saut peut effectivement être enlevée de nos calculs.

Concernant la soustraction, nous avons réalisé un test pour vérifier l'impact de soustraction de nombre entier lors de l'exécution d'un code ne réalisant que des opérations sur des nombres flottants (voir extrait 3.8). En mesurant la performance de ce code, nous avons observé que 4 soustractions sur des nombres entiers n'impactent pas la performance de la boucle. Au-delà de 4, le processeur a besoin de cycles supplémentaires pour les exécuter. Ainsi les deux instructions de gestion de boucles peuvent être ignorées dans le calcul. Pour réduire l'impact de ces deux instructions sur le résultat donnée par `perf`, nous avons implémenté une option de déroulement de boucle (*unrolling*). En utilisant l'option `-U 10`, le benchmark généré déroule la boucle 10 fois permettant de réduire l'impact des instructions de la gestion de boucle (incrémentation et test). Grâce à cette option, nous avons pu valider l'IPC calculé par notre benchmark avec celui mesuré par `perf` (2 instructions par cycle).

```
cycleInStart = rdtsc();
__asm__ ("
    "myBench: "
        "vaddpd %%zmm0, %%zmm1, %%zmm2; "
        "vaddpd %%zmm0, %%zmm1, %%zmm3; "
        "vmulpd %%zmm0, %%zmm1, %%zmm4; "
        "vmulpd %%zmm0, %%zmm1, %%zmm5; "
        "sub  $0x1, %%ebx;" //soustraction artificielle
        "sub  $0x1, %%ebx;" //soustraction artificielle
        "sub  $0x1, %%ebx;" //soustraction artificielle
    "sub  $0x1, %%eax;"
    "jnz  myBench;"   : "=r" (instructions_executed) : "a" (NB_lOOP_IN));
cycleInEnd = rdtsc();
```

Extrait 3.8 – En ajoutant jusqu'à 3 soustractions, nous avons pu vérifier que la soustraction utilisée pour la gestion de la boucle n'avait aucun impact sur la performance du benchmark. En effet, la soustraction sur un nombre entier n'utilise pas la FPU.

Validation des FLOPS. Afin de valider le nombre de calculs réalisés, deux méthodes ont été employées. La première méthode consiste à utiliser d'un outil développé en interne appelé `mygflops`. Cet outil affiche le nombre d'opérations exécutées en consultant les compteurs matériels de chaque coeur. Le résultat sépare les différentes tailles d'instructions vectorielles utilisées. La commande, le résultat du benchmark et celui de `mygflops` sont présentés dans l'extrait 3.9.

```
1  ./kg -W 128 -O aamm -P double -U 10 -S 10 -L 90000000
2  ...
3  ------------------------- FLOP SUMMARY -----------------------------
4   PRECISION      FLOP/cycle          FLOP/second
5      Single               0                    0
6      Double            4.00             1.06e+10
7  ------------------------------------------------------------------
8  ...
9
10 ********************* MYGFLOPS **********************
11
12 Single-precision SSE/AVX :            0.000000 GFlop/s --  0.0% of Flops
13
14        0.0%  32-bit SSE/AVX instructions (0.0%)
15        0.0% 128-bit SSE/AVX instructions (0.0%)
16        0.0% 256-bit AVX instructions     (0.0%)
17        0.0% 512-bit AVX instructions     (0.0%)
18
19 Double-precision SSE/AVX :            10.572521 GFlop/s -- 100.0% of Flops
20
21        0.0%  64-bit SSE/AVX instructions (   0.0%)
22      100.0% 128-bit SSE/AVX instructions (100.0%)
23        0.0% 256-bit AVX instructions     (   0.0%)
24        0.0% 512-bit AVX instructions     (   0.0%)
```

Extrait 3.9 – L'utilisation de l'outil `mygflops`, développé par notre équipe, a permis de valider les résultats donnés par notre benchmark (*ligne 6*). L'outil `mygflops` utilise les compteurs matériels pour mesurer les différentes instructions de calculs réalisées et permet de valider nos résultats (*ligne 19*).

Malheureusement, l'outil `mygflops` n'est pas disponible en accès libre pour le reste de la communauté. Nous avons ainsi développé une méthode de validation des résultats interne au benchmark. En fonction des opérations utilisées, les registres sont initialisés avec différentes valeurs significatives. Par exemple, pour vérifier que le bon nombre d'additions a été exécuté, les registres sont initialisés à la valeur 1. À la fin du benchmark, les registres ayant participé au benchmark sont sommés pour vérifier que le bon nombre d'additions a été exécuté. Grâce à cette méthode, nous avons la certitude que les opérations sont réellement exécutées par le processeur et qu'aucune optimisation lui permettant d'en éviter n'est possible ou une erreur de logique.

3.3.2.5 Mesure de la fréquence

Plus un processeur utilise une fréquence élevée, plus sa consommation électrique est élevée. Pour cette raison, Intel a adopté différents niveaux de fréquences pour ses processeurs. Cela permet d'augmenter les performances en cas de besoin et de limiter la consommation d'énergie si le processeur n'a pas besoin d'être pleinement utilisé. Les instructions les plus complexes (instructions vectorielles) nécessite l'utilisation de plus de transistors que les instructions sca-

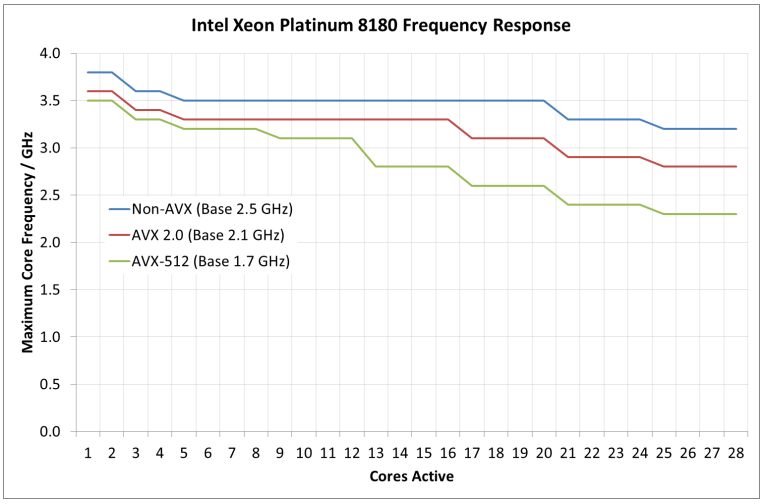


FIGURE 3.19 – La fréquence soutenable par le processeur (axe des ordonnées) varie en fonction du nombre de coeur utilisés (axes des abscisses) et du type d'instructions exécutées : scalaire (en bleu), AVX 2.0 (en rouge) ou AVX 512 (en vert) [7].

laires. L'énergie dissipée est alors plus élevée, empêchant le processeur d'utiliser sa fréquence maximale pour l'exécution de telles instructions (voir figure 3.19).

L'instruction `rdtsc` est utilisée pour lire le compteur matériel mesurant le nombre de *tics* de l'horloge depuis la dernière réinitialisation du processeur [Coo97]. La fréquence à laquelle ce compteur est incrémenté ne varie pas et correspond à la *fréquence de base* du processeur. Cette fréquence est indépendante de la fréquence d'horloge réelle, qui peut varier. Si les fréquences des processeurs devaient varier, les mesures effectuées avec `rdtsc` seraient erronées. C'est pourquoi nous conseillons de fixer la fréquence du processeur avant l'exécution du micro-benchmark. Dans le cas d'une fréquence fixe différente de la fréquence de base, nous avons mis en place une vérification qui calculera ensuite cette fréquence, et ajustera les résultats mesurés par `rdtsc` comme le calcul de l'IPC. Cette fonctionnalité de vérification de la fréquence et de la correction de résultat est utilisable avec l'option `-F true`. Afin d'apporter les corrections voulues, nous avons besoin de mesurer la fréquence de base du processeur ainsi que la fréquence utilisée pour l'exécution.

Mesurer la fréquence de base. Nous avons développé un code permettant de mesurer la fréquence de base. Il utilise la fonction `sleep()` qui attend un certain nombre de microsecondes, ainsi que l'instruction `rdtsc`. Ainsi nous pouvons calculer la fréquence avec le rapport $cycleSpent/timeSpent$ comme indiqué sur l'extrait 3.10.

7. Source du graphique : https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc

```
timeStart = mygettime();
cycleInStart = rdtsc();
    usleep(10000);
cycleInEnd = rdtsc();
timeEnd = mygettime();
cycleSpent = (cycleInEnd - cycleInStart);
freq_Base = cycleSpent / (1000000000 * (timeEnd - timeStart));
```

Extrait 3.10 – Code utilisé pour mesurer la fréquence de base du processeur. La fonction `mygettime()` utilise la fonction `gettimeofday()` [8] permet de mesurer le temps nécessaire à l'exécution du code.

Mesurer la fréquence réelle. Pour pouvoir ajuster les résultats affichés par le benchmark, il est nécessaire de connaître la fréquence à laquelle le processeur est capable de fonctionner. Cette fréquence peut varier soit avec l'utilisation du mode *turbo* soit en limitant la fréquence de façon logicielle. Pour réaliser cette mesure, nous sommes partis du constat que tous les processeurs modernes sont capables d'exécuter une soustraction sur un registre par cycle d'horloge.

```
cycleInStart = rdtsc();
__asm__ ("aloop: "
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "jnz aloop" : : "a" (80000000UL)
);
cycleInEnd = rdtsc();
cycleSpent = (cycleInEnd - cycleInStart);
```

Extrait 3.11 – Code utilisé pour mesurer la fréquence réelle du processeur.

Dans l'extrait 3.11, le registre `%eax` est initialisé à `80000000`. Grâce à notre hypothèse, il est possible de prévoir que le processeur réalisera ces `80000000` soustractions en autant de cycles. Nous mesurons le nombre de cycles nécessaire à l'exécution du code à l'aide de l'instruction `rdtsc`. Ce nombre correspond au nombre de cycle à la fréquence de base du processeur. En faisant le rapport entre le nombre de cycles mesurés et celui attendu par notre hypothèse, il est possible de déterminer si le processeur utilise une fréquence plus ou moins rapide que sa fréquence de base. Nous calculons ainsi l'IPC de cette boucle en faisant le rapport $\frac{80000000}{cycleSpent}$.

- `IPC == 1` : Le processeur utilise sa fréquence de base
- `IPC < 1` : Le processeur utilise une fréquence inférieure à sa fréquence de base.
- `IPC > 1` : Le processeur est capable d'exécuter des instructions à une fréquence plus élevée que sa fréquence de base (turbo).

8. gettimeofday(2) - http://man7.org/linux/man-pages/man2/gettimeofday.2.html

Ajuster les résultats. En connaissant la fréquence de base et la fréquence accessible par le processeur, les résultats donnés par le benchmark peuvent être ajustés. Nous avons utilisé un script externe pour verrouiller la fréquence du processeur Intel Xeon 6150 à 2,00 GHz. Ce processeur a une fréquence de base de 2,70 GHz. Le code présenté ci dessus nous a permis de mesurer un IPC égal à 0,742. Cela signifie que le processeur utilise une fréquence égale à 74,2% de sa fréquence de base, soit 2,00 GHz. Ce dernier test nous a permis de valider la méthodologie basée sur `rdtsc` pour mesurer la fréquence réelle du processeur. Ainsi, nous pouvons ajuster les résultats donnés par notre outil grâce à l'option `--frequency true`.

3.3.3 Résultats

À la suite du développement du générateur de kernels, plusieurs expérimentations ont pu être réalisées. Dans cette section, nous présentons les principaux résultats obtenus lors de la caractérisation de processeurs Intel Skylake.

3.3.3.1 Vérifier les performances théoriques

Le `Kernel Generator` peut être utilisé pour mesurer les performances maximales atteignables par un processeur. Cette performance mesurée en FLOPS est notée $\texttt{FLOPS}_{\texttt{peak}}$. Pour illustrer cette utilisation, nous avons étudié les performances du processeur Intel Xeon 6150 cadencé à 2.7 GHz, possédant 18 coeurs et disposant d'un mode Turbo. Les résultats de cette expérimentation sont donnés dans le tableau 3.7.

Comme expliqué dans l'introduction ci-dessus, la fréquence soutenable par un processeur dépend de la complexité des instructions exécutées et du nombre de coeurs utilisés. La performance de calcul théorique varie donc en fonction de chaque configuration `{nombre de coeurs, type d'instructions, Turbo ON/OFF}`. Pour permettre le calcul de $\texttt{FLOPS}_{\texttt{peak}}$, Intel donne dans sa documentation les différentes fréquences supportées par le processeur [9]. Ces valeurs sont reportées dans la 4e ligne du tableau 3.7. Pour un type d'instruction et un nombre de coeur donné, la performance du processeur dépend de l'activation ou non du turbo :

— **Turbo ON** : si le turbo est activé, la documentation du processeur donne la fréquence maximale atteignable par le processeur pour une configuration `{nombre de coeurs, type d'instructions}` donnée. Cette fréquence, appelée *Maximum Core Frequency* (MCF), n'est pas garantie. Elle est seulement atteignable si la température du processeur le permet. En fonction de sa qualité de fabrication (fuite de courant) et de l'efficacité de son système de refroidissement, la fréquence maximale atteignable peut varier de 20%.

— **Turbo OFF** : si le turbo est désactivé, la documentation donne la fréquence minimale que le processeur utilisera pour exécuter les instructions. Cette fréquence, appelée *Base Core Frequency* (BCF), est dite *garantie*, car le processeur n'utilisera pas de fréquence inférieure à celle-ci. La fréquence BCF peut donc être utilisée pour calculer une limite

9. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf

inférieure de la performance du processeur. Si la température et la conception du processeur le permettent, le processeur peut utiliser des fréquences plus élevées. Cependant, comme le turbo est désactivé, la fréquence ne pourra pas dépasser la fréquence de base du processeur (2.7 GHz dans notre exemple).

Le but de cette expérimentation est d'utiliser le `Kernel Generator` pour vérifier la fréquence atteignable par le processeur pour différentes configurations. La fréquence maximale atteignable par le processeur n'étant pas garantie (même lorsque le turbo est activé), seul l'exécution du kernel peut permettre de mesurer la performance maximale atteignable par le processeur. Cette performance, mesurée en FLOPS, est notée $\texttt{FLOPS}_{\texttt{max}}$. Pour chaque type d'instructions (Non-AVX, AVX 2.0 et AVX-512), $\texttt{FLOPS}_{\texttt{max}}$ peut être mesurée en exécutant des instructions FMA. Nous avons utilisé le `Kernel Generator` pour créer trois benchmarks utilisant trois tailles d'instructions (scalaire, 256 et 512 bits) avec la commande suivante :

```
./kg -W {64,256, 512} -O fffffffffffff -P double -U 80 -S 1 -L 90000000
```

Jeu d'instruction	Non-AVX				AVX 2.0				AVX 512			
Turbo	OFF		ON		OFF		ON		OFF		ON	
Nombre de coeurs	1	18	1	18	1	18	1	18	1	18	1	18
Fréquence (GHz) (source Intel)	2.7	2.7	3.7	3.4	2.3	2.3	3.6	3.0	1.9	1.9	3.5	2.5
$\texttt{FLOPS}_{\texttt{peak}}$ (GFLOPS)	10.8	194.4	14.8	244.8	36.8	662.4	57.6	864	60.8	1094	112	1440
$\texttt{FLOPS}_{\texttt{max}}$ (GFLOPS)	10.8	192.6	14.8	244.2	43	772.2	57.3	858.8	86	1430	112	1430
Fréquence calculée (GHz)	2.7	2.7	3.7	3.4	**2.68**	**2.68**	3.58	2.97	**2.68**	**2.48**	3.5	2.48

Tableau 3.7 – Mesure de la performance du processeur Intel 6150 utilisant trois jeux d'instructions (teintes vertes), avec et sans Turbo (teintes jaunes) et en utilisant un seul ou tous les coeurs (teintes violettes). Les données mesurées (en rouge) sont à comparer avec les spécifications techniques données par le constructeur (en gris). Les performances mesurées peuvent être supérieures aux performances théoriques ($\texttt{FLOPS}_{\texttt{max}} > \texttt{FLOPS}_{\texttt{peak}}$) lorsque le processeur est capable de soutenir une fréquence plus élevée (en bleu) que celle annoncée par le constructeur.

La documentation du fabricant indique que le processeur étudié est capable d'exécuter deux instructions par cycle, quelle que soit la taille des instructions utilisées. Ces instructions nécessitant plus ou moins de transistors pour être exécutées (taille, nombre de coeur), le processeur doit adapter sa fréquence pour ne pas surchauffer. Le `Kernel Generator` mesure le nombre d'instructions par cycle, ce qui nous permet de vérifier que deux instructions FMA sont bien exécutées chaque cycle. Le benchmark permet de vérifier un IPC de 2 et mesure la performance $\texttt{FLOPS}_{\texttt{max}}$ en GFLOPS du code. Grâce à ces deux informations, il est possible de calculer la fréquence réelle qu'utilise le processeur (dernière ligne du tableau 3.7). Le calcul de la performance maximale théorique utilisant la fréquence garantie par Intel, il est possible d'obtenir $\texttt{FLOPS}_{\texttt{max}}$ supérieur à $\texttt{FLOPS}_{\texttt{peak}}$. Nous remarquons ainsi, quatre fréquences supérieures à celles annoncées par la documentation (en bleu dans le tableau 3.7). En effet, Intel communique la fréquence minimale garantie pour chaque configuration texttt{nombre de coeurs, type d'instructions, Turbo

ON/OFF}. Cette fréquence minimale est garantie pour tous les processeurs d'un même modèle (même SKU). La qualité de fabrication du processeur peut lui permettre d'atteindre des fréquences plus élevées que celle-ci, comme indiqué en bleu dans le tableau 3.7. Pour étudier l'évolution de la fréquence et de la température du processeur, nous utilisons un outil développé en interne par HPE. La figure 3.20 montre que le processeur est capable d'utiliser une fréquence de 2.7 GHz, alors que la fréquence minimale garantie est de 2.3 GHz. Le benchmark a été exécuté pendant trente minutes. Le bon système de refroidissement utilisé empêche le processeur de dépasser sa puissance TDP (Thermal Design Power) et conserve cette fréquence durant toute l'exécution. La puissance TDP est mesurée en watt et exprime la quantité de chaleur dégagée par le processeur lorsqu'il est en charge. Le TDP permet au constructeur de systèmes de refroidissement de calibrer le matériel nécessaire pour refroidir un processeur.

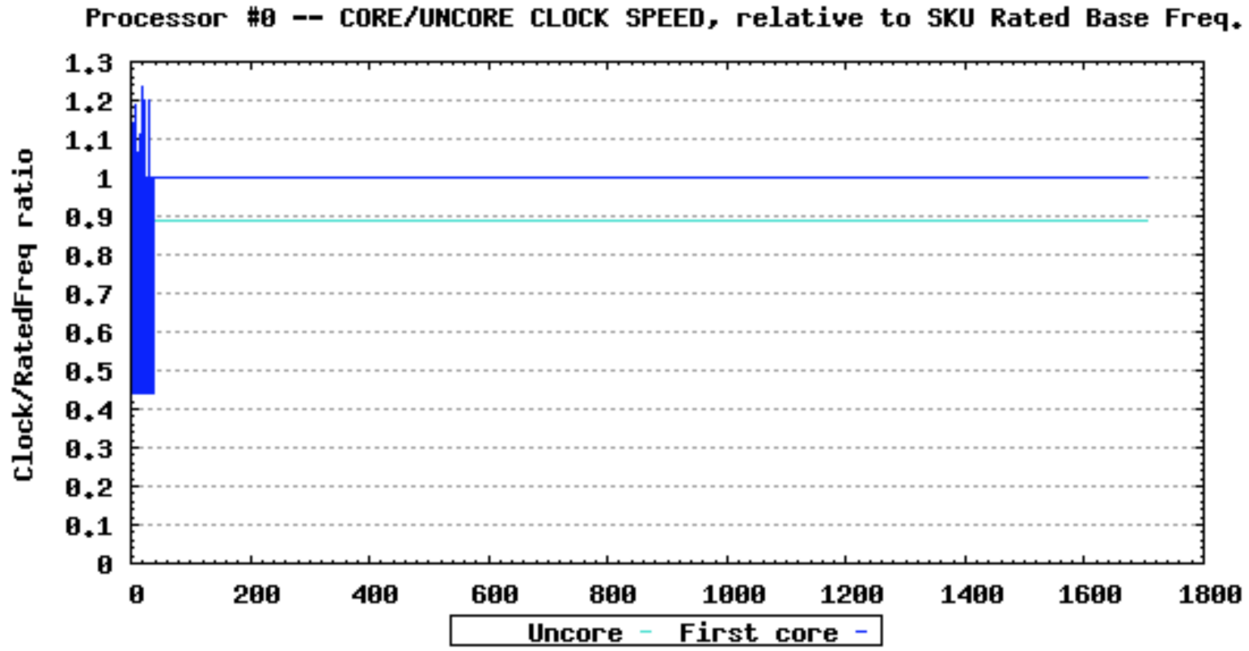
3.3.3.2 Caractérisation de la micro-architecture Haswell

Lors de l'utilisation d'une nouvelle plateforme, l'utilisateur peut utiliser le `Kernel Generator` pour caractériser la microarchitecture et trouver ses points forts ou points faibles. Lors de l'arrivée des processeurs Intel de génération Haswell en 2013, certains codes ont connu des baisses de performances malgré l'utilisation d'une architecture plus récente. Nous avons utilisé le `Kernel Generator` pour caractériser les performances d'exécution des additions et des multiplications avec la commande suivante : `./kg -P double -W 64 -O mmmmm` permettant de générer le benchmark suivant :

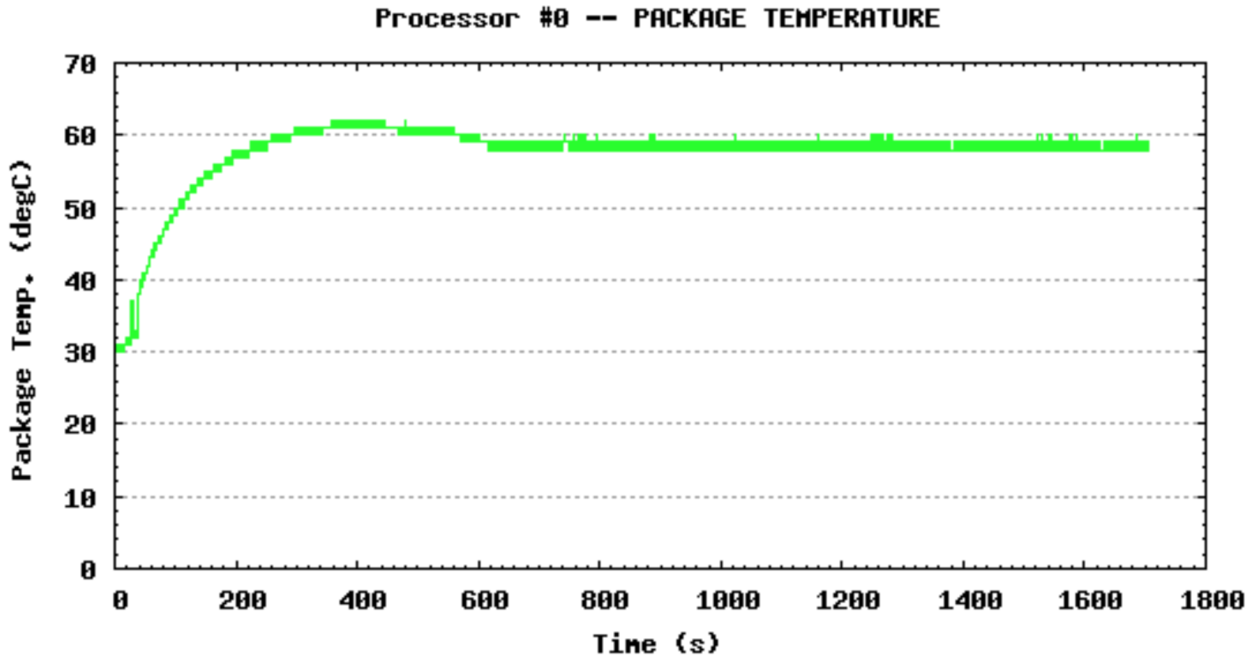
```
1  for (i = 0; i < NB_lOOP; i++) {
2  timeStart = mygettime();
3  cycleInStart = rdtsc();
4  __asm__ (""
5        "myBench:"
6         "vmulsd %%xmm0, %%xmm1, %%xmm2; "
7         "vmulsd %%xmm0, %%xmm1, %%xmm3; "
8         "vmulsd %%xmm0, %%xmm1, %%xmm4; "
9         "vmulsd %%xmm0, %%xmm1, %%xmm5; "
10        "vmulsd %%xmm0, %%xmm1, %%xmm6; "
11       "sub  $0x1, %%eax;"
12       "jnz  myBench;"     :: "a" (NB_lOOP_IN));
13 cycleInEnd = rdtsc();
14 timeEnd = mygettime();
15 cycle_total += (cycleInEnd - cycleInStart);
16 time_total += timeEnd - timeStart;
17 }
```

La microarchitecture Haswell est capable d'exécuter deux multiplications par cycle. Cependant, l'utilisation du `Kernel Generator` nous a montré que le processeur n'était pas capable d'exécuter des additions au même rythme que les multiplications. Les résultats présentés dans le tableau 3.8 montrent que la microarchitecture Haswell est capable d'exécuter deux multiplications contre une seule addition par cycle.

Bien sûr, cette caractéristique est documentée et en comptant le nombre de ports desti-



(a) Évolution de la fréquence. Un ratio de 1 correspond à la fréquence de base de processeur (2.7 GHz).



(b) Évolution de la température du processeur

FIGURE 3.20 – Évolution de la fréquence et de la température du processeur pour un benchmark AVX 2 exécuté sur 18 coeurs avec le turbo désactivé.

Opération	Nombre d'instructions	Frequence	Temps	IPC
Multiplication	40000000000	2.1	7.71	**2**
Addition	40000000000	2.1	14.43	**1**

Tableau 3.8 – Différence de performance lors de l'exécution d'addition et de multiplication sur une architecture Haswell.

nés aux additions, l'utilisateur du processeur aurait pu en trouver la raison. Cependant la lecture de la documentation de la microarchitecture dépasse le millier de pages et en comprendre les moindres détails est plus difficile que d'utiliser les bons outils. Nous pensons que le `Kernel Generator` peut permettre à n'importe quel utilisateur de rapidement trouver ce genre de caractéristiques. Avant même d'avoir exécuté son application sur une nouvelle plateforme, il peut rapidement se faire une idée de ses performances et trouver ce genre de défauts.

3.3.3.3 Caractérisation de l'exécution dans le désordre

La puissance d'un supercalculateur vient de sa capacité à réaliser des calculs en parallèle. Pour satisfaire la loi d'Amdahl (voir section 2.1.4.1). La tâche des développeurs est de maximiser les zones de codes pouvant profiter des ressources parallèles des architectures. Le principal frein à l'utilisation du parallélisme vient de zones de codes dites séquentielles dont les instructions doivent être exécutées à la suite les unes des autres. Cela peut être dû à la dépendance d'une instruction au résultat de l'instruction précédente. Les performances d'un tel code sont souvent très mauvaises car une seule ressource de calcul (un coeur) peut être utilisée pour leur exécution. Cependant, la nature des algorithmes peut assez fréquemment laisser place à certaines optimisations. Dans le domaine des finances, les algorithmes de Monte-Carlo sont très utilisés. Ces codes ont la particularité d'exposer de longues chaînes de dépendances. En restructurant le code, le programmeur peut espérer profiter de l'exécution dans le désordre (voir Annexe A.2.3.5). Cela nécessite cependant d'apporter suffisamment d'instructions indépendantes au processeur pour qu'il puisse les exécuter en parallèle. Le processeur est capable d'exécuter plusieurs chaînes indépendantes les unes des autres. La performance d'une telle plateforme dépend alors de sa capacité à en exécuter plusieurs en parallèle. Grâce à l'option `--dependency N` du `Kernel Generator`, le benchmark peut être utilisé pour caractériser cette fonctionnalité matérielle en générant N chaînes indépendantes. En utilisant une dépendance de 1, chaque instruction a besoin du résultat de l'instruction précédente (voir extrait 3.12). Dans ce cas-là, aucune parallélisation n'est possible pour le processeur. L'exécution de ce code atteint un IPC de 0.25. Le processeur a besoin de 4 cycles d'horloge pour exécuter une instruction. Le faible IPC vient de la nécessité d'attendre que le résultat de l'opération précédente soit disponible pour pouvoir commencer à être exécutée. Sur le processeur utilisé, l'exécution d'une multiplication AVX-512 est de 4 cycles.

```
                          source        cible
"myBench: "
      "vmulpd %%zmm0, %%zmm6, %%zmm2; "
      "vmulpd %%zmm0, %%zmm7, %%zmm3; "
      "vmulpd %%zmm0, %%zmm8, %%zmm4; "
      "vmulpd %%zmm0, %%zmm9, %%zmm5; "
      "vmulpd %%zmm0, %%zmm2, %%zmm6; "
      "vmulpd %%zmm0, %%zmm3, %%zmm7; "
      "vmulpd %%zmm0, %%zmm4, %%zmm8; "
      "vmulpd %%zmm0, %%zmm5, %%zmm9; "
"sub  $0x1, %%eax;"
"jnz  myBench;"       : "=r"
```

FIGURE 3.21 – Code généré par la commande `/kg -W 512 -P double -O mmmmmmmm -D 4`. Les quatre chaînes de dépendance sont représentées par les quatre couleurs utilisées.

```
1 "myBench: "
2   "vmulpd %%zmm0, %%zmm6, %%zmm2; "
3   "vmulpd %%zmm0, %%zmm2, %%zmm3; "
4   "vmulpd %%zmm0, %%zmm3, %%zmm4; "
5   "vmulpd %%zmm0, %%zmm4, %%zmm5; "
6   "vmulpd %%zmm0, %%zmm5, %%zmm6; "
7 "sub  $0x1, %%eax;"
8 "jnz  myBench;"
```

Extrait 3.12 – Code généré par la commande `/kg -W 512 -P double -O mmmmm -D 1`. Chaque instruction utilise le résultat produit par l'instruction précédente.

Pour évaluer la capacité du processeur à exécuter plusieurs chaînes indépendantes, différents nombres de chaînes ont pu être générées grâce à l'option `--dependecy N`. Le code généré pour 4 chaînes est présenté sur la figure 3.21. En faisant varier le nombre de chaînes indépendantes, les résultats présentés dans le tableau 3.9 ont été obtenus. Grâce au tampon d'instructions de système d'exécution dans le désordre, le processeur est capable de commencer l'exécution de plusieurs chaînes indépendantes simultanément. Nous avons pu valider que le processeur est capable d'exécuter au moins 8 chaînes d'instructions indépendantes. Le processeur peut exécuter une multiplication par cycle par pipeline (2 au total). La latence d'une multiplication AVX-512 étant de 4 cycles, le processeur a besoin de 8 chaînes indépendantes pour utiliser la totalité de la puissance du processeur. Cette caractéristique du processeur doit être connue par le programmeur pour transformer son code et obtenir le maximum de performance du processeur. Grâce au `Kernel Generator`, les nouvelles architectures peuvent être testées pour caractériser ces performances et prévoir leur performance pour des codes pouvant utiliser des chaînes de calculs indépendantes comme la résolution de polynômes par exemple.

Nombre de chaînes	1	2	3	4	5	6	7	8	9	10
IPC	0.25	0.50	0.75	1	1.25	1.50	1.75	2	2	2

Tableau 3.9 – Mesure du nombre d'instruction par cycle pour un nombre de chaîne de dépendance N, dans un benchmark généré à l'aide de la commande `/kg -W 512 -P double -O mmmmmmmm -D N`. Au moins 8 chaînes d'instructions indépendantes doivent être présentées au processeur pour l'utiliser au maximum de ses performances (IPC égal à 2).

3.3.3.4 Caractérisation de la FPU de deux processeurs Skylake

La dernière génération de processeur Intel Skylake est répartie en quatre gammes (Bronze, Silver, Gold et Platinium). Les fonctionnalités et les performances des processeurs des différentes gammes étant différentes (voir tableau 3.10), il est important pour notre équipe avant-vente d'en connaître les caractéristiques et les performances pour adapter les configurations des serveurs aux demandes des clients. Les processeurs des gammes Bronze ou Silver sont rarement choisis pour répondre à un appel d'offres compte tenu de leurs caractéristiques plus faibles (une FPU au lieu de deux, nombre plus faible de coeurs). Les processeurs haut de gamme (Gold et Platinium) possèdent généralement plus de coeurs, et sont capables d'utiliser des fréquences plus élevées que les processeurs d'entrée de gamme (Bronze et Silver). Bien sûr, comme ces processeurs coûtent plus cher à l'achat, un client choisira un processeur en fonction de son budget, de sa consommation électrique ou du profil de son application.

Gamme - référence CPU	Bronze : 31xx	Silver : 41xx	Gold : 51xx	Gold : 61xx	Platinium : 81xx
Nb. canaux mémoire et vitesse	6-ch@2133 Ghz	6-ch**@2400**	6-ch@2400	6-ch**@2666**	6-ch@2666
Lien UPI (scalabilité)	2 (2S-2UPI)	2 (2S-2UPI)	2 (4S-2UPI)	3 (2S-3UPI)	3 (8S-3UPI)
Débit UPI	9.6 GT/s	9.6 GT/s	10.4 GT/s	10.4 GT/s	10.4 GT/s
HyperThreading [Mar+02]	NON	**OUI**	OUI	OUI	OUI
FMA-512 FPU	1	1	1	**2**	2

Tableau 3.10 – Principales différences entre les gammes de processeurs Intel de génération Skylake.

Pour caractériser la performance crête des processeurs $\texttt{FLOPS}_{\texttt{max}}$, notre équipe de benchmark utilise des codes tels que HPL. Pour obtenir la meilleure performance atteignable, le benchmark HPL est compilé pour utiliser les instructions vectorielles les plus grandes supportées par le processeur étudié (ici des instructions AVX-512). Les résultats du benchmark Linpack donnés dans le tableau 3.11, montrent que les processeurs les plus performants sont ceux appartenant aux gammes Gold et Platinium. Cette différence de performance peut être expliquée par la présence de deux FPU sur les processeurs haut de gamme. Ces deux FPU sont capables d'exécuter chacune une instruction FMA AVX-512 par cycle. Grâce au `Kernel Generator`, cette caractéristique a pu être vérifiée (voir tableau 3.11).

Processeur (Nb. FPU)	Silver : 4110 (1)	Gold : 5117 (1)	Gold : 6130 (2)	Platinium : 8160 (2)
Instruction AVX-512 FMA par cycle	1	1	2	2
Benchmark HPL (GFLOPS)	173	179	349	354

Tableau 3.11 – Pour différents processeurs, le nombre d'instructions FMA AVX-512 pouvant être exécutée chaque cycle a été mesuré à l'aide du `Kernel Generator`. En fonction du nombre de FPU présent sur un coeur (1 ou 2), le nombre d'instructions exécutées varie (1 ou 2). La performance $\texttt{FLOPS}_{\texttt{max}}$ du benchmark HPL mesurée GFLOPS varie de la même façon suivant la gamme du processeur utilisé. Afin de pouvoir comparer les différentes gammes de processeurs, la configuration suivante a été appliquée à chaque processeur : désactivation de l'*hyperthreading*, fréquence limitée à 1.5 GHz, utilisation de 8 coeurs.

Afin d'estimer l'impact d'une FPU manquante sur la performance d'une application réelle, nous avons utilisé une application de CFD (*Computational Fluid Dynamics*) typique. L'application utilisée n'exécute que des instructions vectorielles de 256 bits (AVX-2). Nous nous attendons alors à obtenir des performances différentes entre un processeur de gamme Silver et de gamme Gold pour une application qui n'est pas limitée par la bande passante mémoire. Étonnamment, les résultats obtenus sur ces deux plateformes sont très proches. Pourtant, nous avons bien montré que les processeurs de gamme supérieure possèdent une FPU de plus et sont donc deux fois plus performants. Pour comprendre ce phénomène, nous avons utilisé le `Kernel Generator` pour générer un kernel de calculs utilisant des instructions vectorielles de 256 bits. La performance mesurée pour le kernel est reportée dans le tableau 3.12.

	Silver : 4110	Gold : 6130
IPC	2	2
GFLOP/s	2.38e+10	2.37e+10

Tableau 3.12 – Mesure de la performance de processeur de deux gammes différentes à l'aide du `Kernel Generator` et de la commande suivante : `/kg -P double -W 256 -O ffffffff`. Alors que le processeur de la gamme Gold possède une FPU de plus, il obtient des performances similaires à celles du processeur de gamme Silver.

Bien que le processeur Intel Xeon Silver 4110 ne possède qu'une seule FPU, il est capable d'exécuter deux instructions AVX-2 par cycle. Cette caractéristique peut être retrouvée dans la documentation du processeur [10]. Les FPU des processeurs Skylake d'entrée de gamme fusionnent deux ports de 256 bits pour former la FPU 512-bits. Cependant, lorsque des instructions de 256 bits sont exécutées, le processeur peut utiliser les deux ports indépendamment pour exécuter deux instructions de 256 bits. La FPU supplémentaire présente sur les processeurs haut de gamme est une FPU 512 bits qui ne permet pas d'utiliser cette caractéristique et qui pourrait permettre d'exécuter (en théorie) quatre instructions par cycle. Afin de mieux comprendre

10. source : https://en.wikichip.org/wiki/intel/microarchitectures/skylake#Execution_engine_2

comment cette fusion d'instructions fonctionnait, nous avons utilisé le `Kernel Generator` pour générer des kernels possédant des instructions vectorielles de taille différentes. Les résultats des différentes expérimentations sont reportés dans le tableau 3.13. Chaque cycle, la FPU est capable d'exécuter différentes combinaisons d'instructions :

— Une instruction AVX 512 bits,
— Deux instructions AVX 256 bits,
— Deux instructions AVX 128 bits,
— Deux instructions scalaires,
— Toute combinaison de deux instructions dont la taille agrégée ne dépasse pas 512 bits.

Gamme	Silver	Gold	Gold	Platinium
Ref. processeur Intel Skylake	4110	5117	6130	8160
Nombre de FPU	1	1	2	2
128 + scalaire	**2**	**2**	2	2
256 + scalaire	**2**	**2**	2	2
256 + 128	**2**	**2**	2	2
256	**2**	**2**	2	2
512 + scalaire	1	1	2	2
512 + 128	1	1	2	2
512 + 256	1	1	2	2

Tableau 3.13 – Mesure du nombre d'instruction exécutées chaque cycle pour différents processeurs. La FPU des processeurs d'entrée de gamme est capable de fusionner certaines instructions (en gras) pour les exécuter en un seul cycle.

Ainsi, le processeur haut de gamme bénéficie des deux FPU lorsque le code est capable d'utiliser des instructions vectorielles de 512 bits. Il est fréquent que les applications n'y parviennent pas (mauvaise vectorisation du code, problème du compilateur, dépendances) et utilisent des instructions vectorielles plus petites. Les processeurs d'entrée de gamme obtiennent alors des performances rigoureusement égales. Évidement, la FPU n'est pas la seule responsable de la performance d'une application, le tableau 3.10 montre bien que d'autres caractéristiques diffèrent telles que le nombre de liens UPI ou la possibilité d'utiliser l'hyperthreading. Le but du `Kernel Generator` est de caractériser une architecture pour le besoin d'une application. Grâce aux caractéristiques découvertes suite à cette expérimentation, plusieurs réponses à des offres d'appels ont été réalisées avec des processeurs d'entrée de gamme. Grâce à cette caractéristique, les applications utilisant des instructions AVX-2 peuvent obtenir des performances assez proches sur des processeurs coûtant beaucoup moins cher.

3.3.4 Conclusion

L'outil `Kernel Generator` est un `Kernel Generator` qui permet de mesurer la performance maximale ($\texttt{FLOPS}_{\texttt{max}}$) d'un processeur pour un type d'instruction vectorielle utilisant différentes configurations (type d'instructions, dépendances...). Le `Kernel Generator` assembleur est un outil très précis pour la caractérisation des FPU. Le principal avantage de l'utilisation de l'assembleur est d'éliminer les optimisations du compilateur. Le benchmark doit assurer à l'utilisateur qu'il mesure bien la performance du code qu'il a choisi de générer et qu'il n'a pas été modifié durant la compilation. Il nous a permis d'atteindre des performances souvent égales aux performances théoriques ($\texttt{FLOPS}_{\texttt{peak}}$). En utilisant le générateur, le programmeur peut être amené à découvrir des particularités de la microarchitecture (comme celle de l'exécution dans le désordre). En comprenant précisément le fonctionnement de la FPU, il pourra même trouver des optimisations pour sa propre application.

Pour le moment, seul l'ISA x86 est supporté, mais l'outil a été codé de façon à faciliter l'ajout d'une nouvelle ISA. Nous avons choisi de commencer à le développer pour des processeurs dont nous connaissons bien le comportement pour valider son bon fonctionnement. L'exemple de la FPU du processeur Intel Xeon 4110, nous a permis de montrer que même sur des architectures que nous connaissons bien, certaines spécificités nous échappent encore. L'utilisation du générateur permet d'en comprendre toutes les particularités.

Perspectives La suite du travail comprend la fin du développement de l'option permettant de mélanger différentes tailles d'instructions vectorielles dans le même kernel. Nous souhaitons développer une option qui permettra de générer des instructions des déplacements mémoires afin vérifier qu'elles ne gênent pas l'exécution des instructions de calcul. Enfin, pour mieux caractériser les plateformes pour certaines applications comme en cryptographie, nous allons permettre l'utilisation d'autres instructions (rotation, décalage, etc).

3.4 Monitoring du bus mémoire

Cette section présente le développement de notre outil `YAMB` (*Yet Another Memory Bandwidth profiling tool*). `YAMB` mesure l'activité des différents contrôleurs mémoire (en lecture et en écriture) ainsi que le nombre d'évènements miss dans le dernier niveau de cache (LLC). L'évolution du trafic mémoire peut ensuite être affichée sous forme de graphique (voir figure 3.22). L'outil propose l'utilisation d'une bibliothèque, permettant d'annoter facilement certaines zones du code pour faciliter la lecture du graphique.

FIGURE 3.22 – Exemple d'utilisation de l'outil `YAMB` pour suivre l'activité du bus mémoire lors de l'exécution du benchmark `STREAM` [McC95] pour deux itérations des 4 noyaux de calculs : `copy`, `scale`, `add` et `triadd`.

3.4.1 Introduction

3.4.1.1 Motivation et objectifs

La majorité des processeurs utilisés en HPC utilisent une architecture Von Neuman (voir Annexe A.2.1.2). Pour la majorité des applications de calculs intensifs [Dre07], le système mémoire de ces architectures est le principal responsable de la limitation des performances. Nous observons une augmentation du nombre de coeurs sur les processeurs de dernière génération sans réelle évolution de la performance de cette ressource critique. Cette disparité résulte en une

augmentation d'accès concurrents au bus mémoire, limitant la performance des applications. De plus, ce bus étant une ressource partagée par les coeurs d'un même processeur, la mauvaise utilisation de celui-ci par un coeur affecte la performance des autres.

La majorité des applications HPC étant limitée par la performance du bus mémoire, il est primordial de valider l'utilisation optimale du bus mémoire. Pour cela, il est nécessaire de vérifier qu'il est utilisé à son débit maximal (saturation), mais aussi de l'efficacité des transferts réalisés (transferts de données utilisées par le processeur). Pour réaliser ces vérifications, il n'est pas envisageable de mesure le débit mémoire moyen lors de l'exécution de l'application ou même d'un kernel. En effet, une application peut être limitée par la performance du bus, sans que celui-ci soit saturé sur la totalité de l'exécution du code. Par exemple, si les accès sont tous réalisés au même moment, le bus peut être saturé pendant un court instant et être ensuite inutilisé. Sur la totalité de l'exécution, on obtiendra une mesure qui indiquera que le bus n'est pas saturé alors qu'il s'agit bien du goulot d'étranglement des performances de l'application.

L'objectif de l'outil présenté dans cette section est de répondre aux deux prérogatives indiquées ci-dessus : saturation du bus et utilisation effective. Pour y répondre, l'utilisateur doit posséder un outil lui permettant de suivre son activité avec une résolution fine (de l'ordre de la milliseconde). L'outil en question doit pouvoir distinguer les accès réalisés en lecture et en écriture pour pouvoir s'assurer de l'utilisation effective du bus. Nous expliquons comment le rapport entre les quantités de données en lecture et en écriture peut être utilisé pour mesurer la performance d'un code dans la section 4.4.4.

3.4.1.2 Verrous et état de l'art

La principale difficulté pour le développement d'un tel outil vient de l'utilisation des compteurs matériels permettant de compter les évènements relatifs aux accès mémoires. Ces compteurs permettent de compter les évènements se produisant sur un processeur et sont présentés dans l'Annexe B. Ils sont accessibles par le biais de composants appelés Performance Monitoring Unit (PMU). Un processeur moderne possède généralement au moins deux familles de PMU : une PMU *oncore* situé sur chaque coeur du processeur et une PMU *uncore* située directement sur le processeur. Comme étudié dans la Annexe B.2.2, les compteurs responsables du comptage des évènements liés aux transferts mémoires sont situés sur les PMU *uncore*. Sur des architectures modernes telles que les processeurs Intel Skylake, ces compteurs comptent les accès mémoire réalisés en distinguant la lecture de l'écriture.

Mis à part certaines tâches, comme la gestion des pages, les accès mémoire sont réalisés directement par le contrôleur mémoire. Le système d'exploitation n'a généralement pas connaissance de l'évolution de ces accès, contrairement à d'autres ressources comme les *entrées/sorties* (I/O) pour lesquelles le système d'exploitation joue le rôle d'intermédiaire avec l'application. Ainsi, comme les compteurs ne sont associés à aucun coeur, il est impossible de faire correspondre un accès mémoire au coeur et donc au processus qui en est responsable. Pour des outils nécessitant une grande précision [Lar16], l'utilisation de ces compteurs n'est donc pas possible. Notre outil a pour objectif d'analyser l'activité d'applications HPC qui utilisent généralement tous les coeurs des processeurs pour la même application. Cette particularité n'est donc pas un verrou

majeur pour notre développement.

La programmation des PMU varie d'une architecture à l'autre, ce qui les rend difficiles à programmer et à maintenir. Que ce soit pour le *core* ou le *uncore*, ces méthodes sont assez complexes à utiliser et à maintenir. C'est pourquoi il est courant d'utiliser des interfaces de haut niveau telles que `PAPI` [Bro+00] ou `perf` qui permettent de ne plus dépendre de l'implémentation des compteurs à bas niveau. Ces interfaces ont permis le développement de nombreux outils (voir section 2.4.2.3). Intel propose `VTune` [Rei05] et Intel `MBM`[11]. Le premier est un outil propriétaire nécessitant une licence payante. Le deuxième est proposé en libre accès sur le dépôt en ligne d'Intel mais n'est compatible qu'avec des architectures Intel. D'autres outils tels que `TAU` [Lin+00] ou `Extrae` [Rod] présentent de nombreuses informations à l'utilisateur qui rendent difficile la compréhension des résultats. De plus, ce grand nombre d'informations nécessite généralement l'utilisation de plusieurs compteurs matériels qui peuvent ne plus être présents d'une architecture à l'autre. La complexité des outils peut aussi les rendre dépendants de bibliothèques externes non compatibles avec certaines architectures rendant leur utilisation impossible (`MAQAO` [DB05]). D'autres travaux tels que `Memguard` [Yun+13] mesurent le nombre de miss dans le dernier niveau de cache pour en déduire le trafic du bus mémoire. Malheureusement, avec la complexification des architectures et l'utilisation constante des unités de préchargement mémoire, certains transferts mémoires sont réalisés avant qu'un évènement miss ne soit déclenché. Il n'est donc plus possible de mesurer le trafic mémoire avec ces techniques-là.

3.4.2 Développement de YAMB

De nombreux outils et interfaces ont été développés pour accéder aux compteurs matériels avec leurs avantages et leurs inconvénients. Les principales contraintes des outils développés au cours de ce travail de thèse sont la portabilité et l'accès libre aux sources. Après avoir testé les différents outils et interfaces disponibles, nous avons choisi de baser le développement de `YAMB` sur `Perf Events` [Wea13] (voir section 2.4.2.3).

3.4.2.1 Perf Events

Contrairement à d'autres outils tels que `Likwid` [THW10] ou `PAPI` [Bro+00], le système de suivi de performance `Perf Events` fait partie du projet Linux. Son intégration dans le noyau nous assure un maximum de disponibilité dans les environnements de calcul haute performance ainsi qu'un maximum de compatibilité avec les architectures émergentes. La principale difficulté de `Perf Events` vient l'utilisation de l'appel système *perf_event_open* qui permet de programmer les compteurs matériels [SMM17]. S'il permet d'éviter à l'utilisateur d'écrire manuellement les différents bits de configuration des registres, beaucoup de travail reste à faire pour le développeur désireux de profiler ses applications. En effet, Linux supporte de nombreux processeurs possédant différentes versions de PMU, les développeurs du noyau ont donc laissé la programmation bas niveau à l'utilisateur. En conséquence, cet appel système est très complexe

11. Intel MBM - https://github.com/intel/intel-cmt-cat/wiki

à utiliser et ne peut pas être utilisé de manière portable. Les principales difficultés consistent à trouver les événements à compter ou à échantillonner, à configurer tous les paramètres à transmettre à l'appel système et à effectuer plusieurs appels système en fonction du nombre de *threads* de l'application profilée et du nombre de coeurs utilisés. Ces différentes difficultés (programmation, portabilité) ont été les principales motivations du développement d'autres outils tels que `PAPI` [Bro+00], `Intel PCM`[12] ou `NUMAP` [SMM17].

En plus de l'appel système, `Perf Events` fournit un outil accessible depuis l'espace utilisateurs lui permettant de contrôler le profilage. Nommé `perf`, ce programme utilise l'interface noyau pour réaliser des mesures soit en échantillonnage soit en comptage (voir section 2.4.2.1). Comme présenté dans la section 2.4.2.3, *perf* est l'outil de profilage de Linux, et nous espérons que sa large disponibilité rendra notre outil facilement utilisable par le plus grand nombre de personnes. Bien qu'il s'agisse avant tout d'un outil d'espace utilisateur, la commande `perf` fait partie du noyau Linux du point de vue du développement. Faire partie de Linux assure une haute exigence du développement du code ainsi qu'un support au fil des versions du noyau. En raison des permissions limitées possédées par les utilisateurs de supercalculateur, l'outil doit utiliser une interface ne nécessitant pas de droits privilégiés (*root*) pour fonctionner. Du fait du développement de l'interface `perf_event` dans le code noyau, il est possible pour l'administrateur d'autoriser les utilisateurs normaux à accéder aux compteurs. Cela peut être fait en écrivant la valeur 1 dans le fichier `/proc/sys/kernel/perf_event_paranoid`. Lorsque le noyau supporte le nom symbolique des évènements, `perf` est très simple à utiliser. Dans le cas contraire, `Perf Events` offre la possibilité aux utilisateurs expérimentés d'encoder leurs propres évènements.

3.4.2.2 Fonctionnement de l'outil YAMB

La commande `perf` peut être utilisée pour programmer les PMU *uncore* et accéder aux compteurs des contrôleurs mémoires. L'outil `YAMB` utilise cette commande pour configurer les PMU pour qu'elles comptent les transactions en cours sur chaque canal mémoire, en lecture et en écriture. La commande peut aussi être utilisée pour compter le nombre d'évènements miss dans le cache de dernier niveau.

Le lancement de la commande `perf` en arrière-plan et le traitement des données dans un fichier de sortie constituent le coeur de l'outil de profilage `YAMB`. Ensuite, un script peut être utilisé pour dessiner le graphique. L'outil `YAMB` reçoit deux options `--start` et `--stop`. L'utilisation de la première option lance la commande `perf` avec les arguments adéquats en arrière-plan. L'appel du script avec l'option `--stop` s'occupe de retrouver le `PID` du processus de `perf` pour l'arrêter et de sauver les données collectées dans un fichier. Entre ces deux appels, l'utilisateur peut exécuter son application ou attendre un certain temps :

```
1 $ ./yamb.sh --start
2 $ ... sleep | run application ...
3 $ ./yamb.sh --stop
```

12. Intel Performance Counter Monitor - https://software.intel.com/en-us/articles/intel-performance-counter-monitor

Comme le montre l'extrait de code ci-dessus, l'avantage de cet outil réside dans la flexibilité de son utilisation. Il est courant dans le travail d'analyste de vouloir suivre l'exécution d'une application pendant une période donnée. L'exécution de celle-ci pouvant durer plusieurs heures, il est important que notre outil ne nécessite pas d'attendre son exécution complète pour prodiguer ses résultats. Grâce à cette approche, l'utilisateur se connecte à un serveur durant l'exécution de l'application, lance l'outil pendant une période voulue et l'arrête pour analyser les résultats. Cette méthodologie permet de laisser l'application s'exécuter. Le code de l'application pouvant être instrumenté pour annoter le graphique de résultat, il est possible de tracer et d'identifier les parties du programme mesurées par l'outil.

3.4.2.3 Annotation

Pour aider à identifier la zone de code responsable du trafic mémoire, une bibliothèque a été développée. Elle permet d'annoter le code d'une application (`c`, `c++` et `fortran`) à l'aide d'un `label` et d'une `couleur`. La bibliothèque ne contient que la fonction `yamb_annotate_set_event` permettant d'écrire ces informations dans un fichier de journal :

```
int yamb_annotate_set_event(const char * label, const char *couleur){
...
    m_LOG_FILE << time_step << " " << label << " " << couleur << endl;
...
}
```

3.4.2.4 Mesure de l'impact sur la performance

Un défaut majeur des outils de mesure de performance est leur impact sur la performance de l'application étudiée. Nous avons réalisé plusieurs tests avec différentes fréquences d'échantillonnage pour estimer l'impact de `YAMB` sur l'application mesurée. Même lorsque la fréquence la plus rapide soutenable par `perf` est utilisée (100Hz), l'impact sur la performance est très faible (inférieur à 5%). Pour une étude suffisamment précise de l'application, obtenir 100 mesures par seconde semble largement suffisant. Nous considérons que cette baisse de performance est suffisamment faible pour s'assurer que la performance mesurée est proche de la performance réelle.

3.4.3 Utilisation

Cette section présente un exemple d'utilisation de `YAMB` appliqué au benchmark `STREAM` [McC95]. Pour mieux comprendre l'évolution du trafic mémoire, la première étape est d'annoter les différentes parties du code intéressantes. Pour cela, nous utilisons la fonction `yamb_annotate_set_event` pour ajouter une trace sur le graphique lors de chaque début de benchmark. L'analyse peut ensuite être lancée avec les commandes suivantes :

```
$ ./yamb.sh --output log_stream --command ./stream.SKL.192GB
$ python ./format_log.py --data log_stream.perf.mem --annotate log_stream.
    annotate
```

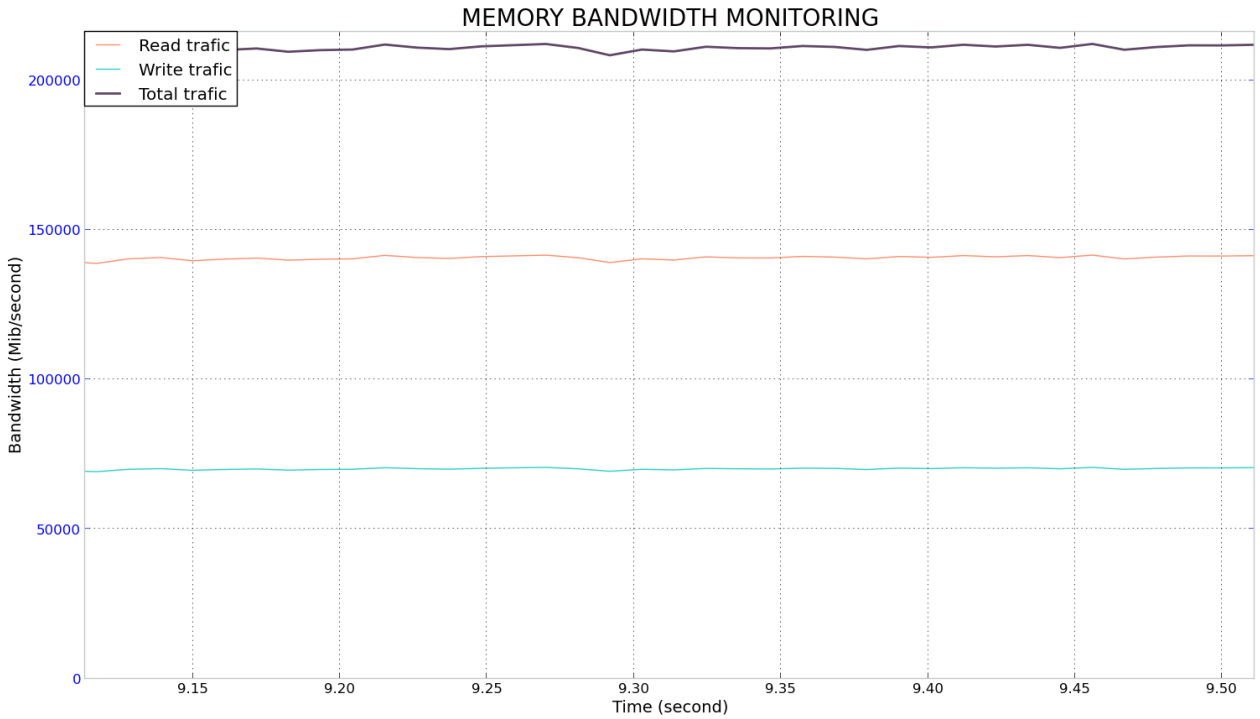


FIGURE 3.23 – Trafic mémoire mesuré en MB/s lors de l'exécution de la fonction `triad` du benchmark `STREAM`.

Le résultat de cette première expérimentation peut être vu sur la figure 3.22. L'utilisation du graphique peut ensuite permettre d'agrandir les parties plus intéressantes comme le noyau de calcul `triad` (voir figure 3.23). Grâce à la distinction entre les accès mémoire réalisés en lecture ou en écriture, nous pouvons remarquer que le ratio de transfert obtenu de deux lectures pour une écriture est bien celui attendu pour le noyau de calcul étudié (voir section 4.4).

Autres utilisations. L'outil `YAMB` est utilisé à plusieurs reprises dans ce manuscrit de thèse. Une utilisation est réalisée avec le benchmark `dml_mem` pour caractériser la capacité du cache de dernier niveau à stocker un jeu de données dont la taille s'approche de celle du cache (section 3.2.3.8). Une autre utilisation est présentée dans le chapitre 4 pour illustrer l'application de la méthodologie et l'utilisation des ratios de lecture et écriture pour s'assurer de l'optimalité d'un code.

3.5 Profil de l'exécution d'instructions

Cette section présente le développement de notre outil `Oprofile++`. Celui-ci fonctionne en deux étapes. La première consiste à suivre les performances du processeur lors de l'exécution d'une application en utilisant les compteurs matériels programmés pour compter le nombre de cycles et le nombre d'instruction exécutées. La seconde étape permet d'extraire les hot spot de l'application. Une fois ces zones de code identifiées, l'outil fait correspondre les évènements enregistrés lors de l'exécution avec les instructions assembleurs qui en sont responsables. Il est ensuite possible de quantifier des opportunités d'amélioration (optimisation du code), mais aussi de prédire la performance de ces boucles en fonction d'une amélioration du matériel ou du logiciel.

```
./oprofile++.sh myapp
    //1\\ START PROFILING
        Starting ... the profiler is now running...

    //2\\ EXECUTION
        Executing ./myapp on CPU #0
        End..

    //3\\ STOP PROFILING

    //4\\ ANALYSIS
    cpu_clk_unhalt|        %|  inst_retired|        %|  app_name
                218    50.67             151    26.77   no-vmlinux
                181    42.00             363    64.36   assembly

    //5\\ KERNEL EXTRACTION

Kernel from the app name (assembly)
hot spot with the symbole name (myBench) which takes 64.36% of the profiling
-------------------------------------------------------------------------------
SUM*4        IPC   CYCLES    INSTS    ADDRESS     ASSEMBLY
-------------------------------------------------------------------------------
 1571 |    0.316       38       12     4018f7     vfmadd231pd %zmm0,%zmm1,%zmm2
 1966 |     1.27      594      756     4018fd     vfmadd231pd %zmm0,%zmm1,%zmm3
 1372 |     1.32      503      664     401903     vfmadd231pd %zmm0,%zmm1,%zmm4
  869 |     1.84      436      804     401909     vfmadd231pd %zmm0,%zmm1,%zmm5
  433 |     1.75      433      758     40190f     sub     $0x1,%eax
    0 |        0        0        0     401912     jne     4018f7 <myBench>
-------------------------------------------------------------------------------
LOOP from 401912 to 4018f7 size= 27
    sum(cycles)= 2004 sum(inst)= 2994 #inst= 6 IPC= 1.49 cycles/LOOP= 4.02
-------------------------------------------------------------------------------
```

Extrait 3.13 – Notre outil Oprofile++ permet d'extraire les instructions des noyaux de calcul à partir du code binaire de l'application et d'y associer des mesures d'évènements.

3.5.1 Introduction

3.5.1.1 Motivations

Pour mener à bien le travail de portage et d'optimisation d'une application, il est important de détecter les hot spots afin qu'ils puissent être optimisés indépendamment les uns des autres [Pop16]. En effet, les hot spots ayant des caractéristiques propres (types d'accès mémoire, pression arithmétique), leur optimisation sera différente. Pour atteindre la performance ultime d'une application, il est ensuite possible de porter chaque hot spot sur un accélérateur adapté.

Le travail d'optimisation des applications est très difficile, notamment sur des applications de calcul haute performance dont le code dépasse souvent les plusieurs milliers de lignes. Il est donc indispensable pour le programmeur d'avoir un outil lui permettant de localiser les hot spots de son application. De nombreuses applications utilisées dans les systèmes d'informations ne présentent pas ce type de profil (un faible pourcentage de ligne de code responsable de la majorité du temps d'exécution de l'application). Par conséquent, leurs profils contiennent un grand nombre de fonctions ou de processus dont le temps d'exécution est passé de façon relativement uniforme [Ama+15]. Ce type de profil est souvent appelé "profil plat" (*flat profil*), car aucune méthode ne domine le temps d'exécution. Dans le calcul haute performance, les hot spots sont souvent facilement identifiables, car la majorité du temps d'exécution se déroule dans ces parties du code.

3.5.1.2 Objectifs

Le programmeur désirant optimiser une application de calcul haute performance doit avoir à sa disposition un outil lui permettant d'obtenir le profil de l'exécution d'une application détaillant le temps passé dans les différentes parties du code. Une fois ces hot spots repérés, l'utilisateur doit connaître précisément comment le code se comporte sur le processeur : type d'instructions utilisées, nombre de cycles par instruction. Utiliser le code source pour cela n'est pas suffisant, car la qualité du code généré par le compilateur peut fortement impacter sa performance (utilisation d'instructions vectorielles, optimisations). Le second objectif de l'outil proposé doit alors présenter le code assembleur ainsi que les évènements associés pour permettre à l'utilisateur d'identifier la cause responsable de sa performance.

3.5.1.3 Travaux existants

De nombreux outils existent pour réaliser l'étude de performance du code. Cependant une grande partie ne respecte pas nos pré-requis exprimés en début de chapitre (voir section 3.1).

`MAQAO` [Bar+10] est un outil développé dans le cadre du projet Mont Blanc [Puz12] en source libre. Il permet de détecter les boucles responsables des hot spots et d'en analyser leur performance. `MAQAO` comprend le développement de l'outil `CQA` [Cha+14] permettant d'évaluer la qualité du code assembleur et de projeter la performance maximale pour une architecture donnée. Il est ainsi possible de connaître les opportunités de vectorisation et des gains potentiels pour les hot spots de l'application. Cette analyse nécessite de connaître beaucoup de spécificités

de l'architecture utilisée et suppose que les données soient directement accessibles dans le cache L1. Cet outil est très orienté sur l'analyse de performance de calculs (FLOPS). Aussi, `MAQAO` permet de réaliser une analyse statique du programme et donne des conseils pour utiliser les drapeaux de compilation adaptés. Nous avons eu des difficultés pour l'utiliser : le dépôt *Git* ne permet pas de s'inscrire facilement pour poser des questions, l'absence de page d'informations *wiki* est aussi un manque. L'outil est dépendant de plusieurs bibliothèques qui n'étaient pas disponibles sur les plateformes à notre disposition.

Intel propose d'utiliser son compilateur `icc` pour annoter automatiquement les fonctions (`-profile-functions`) ou les boucles (`-profile-loops`) d'une application en instrumentant l'entrée et la sortie de ces parties avec l'instruction `rdtsc`[13]. Cette méthode permet d'obtenir une vue d'ensemble de l'utilisation des cycles dans l'application. Il faut cependant posséder le compilateur `icc` (outil propriétaire payant) et cette méthode n'est compatible qu'avec les plateformes du constructeur.

Difficultés et verrous. Le développement d'outil d'analyse de performance est rendu difficile par la faiblesse des compteurs matériels exposée dans la section 2.5. Pour assurer un fonctionnement sur une majorité d'architectures, l'outil doit dépendre du minimum possible d'évènements. Il n'est donc pas possible de pouvoir suivre des évènements complexes bien qu'ils puissent être utiles dans l'analyse de performance. La difficulté vient donc de l'impossibilité de développer un outil complet suffisamment portable pour être utilisé sur une majorité d'architectures.

3.5.2 Développement

L'outil d'analyse de performance `Oprofile`[14] permet de réaliser un échantillonnage des évènements lors de l'exécution de l'application. Cet outil constitue la base du développement de notre version améliorée `Oprofile++`.

3.5.2.1 Étape 1 : Configuration et activation du profiler

La première étape qui suit le lancement de l'outil est la collecte d'informations de l'exécution de l'application étudiée. Pour cela, un premier script est utilisé pour activer le *profiler* (`Oprofile`). Pour faciliter le portage et l'utilisation du profiler sur le plus grand nombre d'architectures, nous avons choisi de ne suivre l'évolution que de deux évènements : le nombre de cycles et le nombre d'instructions exécutées. Dans cette première étape, l'outil `Oprofile++` s'occupe de paramétrer le *profiler* pour compter ces deux évènements à une fréquence qui peut être adaptée pour améliorer la précision de la mesure ou réduire l'impact de l'outil sur les performances de l'application étudiée.

13. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-profile/-function-or-loop-execution-time
14. `OProfile` est un projet open source qui comprend un outil de profilage statistique pour les systèmes Linux - https://oprofile.sourceforge.io/about/

Lorsque l'analyse de performance est terminée, un premier fichier est généré. Il contient les informations de suivi de performance (voir extrait 3.14). Ce fichier contient pour chaque instruction, son adresse mémoire virtuelle et le nombre d'échantillons mesurés pour les deux évènements.

```
vma        samples         %    samples         %   app name     symbol name
00402961        20    0.6553         23    0.3100      app_1     matrix_mult
  00402961       6   30.0000          2    8.6957
  00402964       3   15.0000          3   13.0435
```

Extrait 3.14 – `Oprofile++` exécute l'application et génère un premier fichier contenant l'adresse virtuelle des instructions (*vma*) et le nombre d'évènements correspondant (nombre de cycle, nombre d'instructions).

3.5.2.2 Étape 2 : obtenir les instructions assembleurs et leur adresse mémoire

La première étape a permis d'obtenir le profil de l'application avec la répartition des deux évènements mesurés. Cependant, aucune information n'est donnée concernant le type des instructions responsables, seule leur adresse mémoire est indiquée. Le but de la deuxième étape est de retrouver l'instruction assembleur correspondante. Pour cela, un deuxième fichier est généré à l'aide de la commande `objdump` [15]. Couplé à l'option `d`, cet outil permet de désassembler le binaire de l'application et d'extraire chaque instruction assembleur ainsi que son adresse virtuelle (voir extrait 3.15).

```
0000000000401690 <_init>:
  401694: 48 8b 05 5d 29 20 00   mov    0x20295d(%rip),%rax
```

Extrait 3.15 – L'outil `objdump` permet de désassembler le fichier binaire de l'application.

3.5.2.3 Étape 3 : correspondance des évènements et des instructions

Les instructions sont regroupées par fonction et les fonctions sont triées par le temps passé à leur exécution. Ce tri en ordre décroissant est très utile lors de l'analyse de performance réalisée par le programmeur, car il commence directement par accéder aux instructions des fonctions les plus longues de l'application.

La première contribution de l'outil développé est de faire correspondre les adresses mémoires obtenues lors de l'étape 1 grâce à `Oprofile` avec les instructions contenues dans le fichier généré par `objdump` lors de l'étape 2. Ainsi, nous possédons, dans un même fichier, l'instruction assembleur et les deux compteurs d'évènements associés.

3.5.2.4 Étape 4 : Extraction des noyaux et analyse de performance

L'analyse manuelle du fichier généré précédemment est difficile, même pour un programmeur expérimenté. La deuxième contribution de notre outil `Oprofile++` est le développement

15. `Objdump` permet d'afficher diverses informations sur les fichiers objets sur les systèmes d'exploitation Unix - https://linux.die.net/man/1/objdump

d'un code permettant d'extraire les hot spots et de présenter les principaux à l'utilisateur (extrait 3.16).

```
1  ==========================================
2  _FUNCTION_ANALYSIS_ from the app name (assembly) hot spot from the symbole name
       (myBench) which takes 81.9463% of the profiling
3  ==========================================
4  SUM*4 SUM*3   SUM*2     IPC  CYCLES  INSTS  ADDRESS   ASSEMBLY
5  ------------------------------------------------------------------------------
6   1657  1221     908 |  2.24     211    472   401d79   vaddsd %xmm0,%xmm1,%xmm2
7   1662  1446    1010 |  2.48     697   1731   401d7D   vaddsd %xmm0,%xmm1,%xmm3
8   1593   965     749 |  3.44     313   1077   401d81   vaddsd %xmm0,%xmm1,%xmm4
9   1280  1280     652 |  2.57     436   1122   401d85   vaddsd %xmm0,%xmm1,%xmm5
10   844   844     844 |  2.86     216    618   401d89   vaddsd %xmm0,%xmm1,%xmm6
11   628   628     628 |  3.14     628   1971   401d8D   sub    $0x1,%eax
12 ------------------------------------------------------------------------------
13 LOOP from 401d73 to 401d8D:
14  sum(cycles)= 2501 sum(inst)= 6991 #inst= 7 IPC= 2.79528 cycles/LOOP= 2.50
15 ------------------------------------------------------------------------------
```

Extrait 3.16 – L'outil Oprofile++ permet d'afficher le profil des principaux kernels de l'application.

Les hot spots des applications sont souvent caractérisés par la présence de boucles dans le code. Pour extraire ces noyaux de calcul, notre outil recherche les instructions de saut (*jump*) dans le code en parcourant les instructions dans l'ordre du fichier généré lors de l'étape 2. Ainsi, les premiers résultats affichés sont généralement les boucles critiques de l'application. Lorsqu'une boucle est détectée, elle est affichée, ainsi que les informations suivantes calculées grâce aux deux compteurs d'évènements : le nombre d'instructions, le nombre de cycles, l'IPC et le nombre de cycles par boucle. Avec l'expérience, ces données permettent à un programmeur de comprendre la peformance d'un kernel et d'orienter son travail d'optimisation dans certaines directions.

Sommation d'instruction. Les processeurs utilisés sont tous superscalaires. Ils sont donc capables d'exécuter plusieurs instructions en un seul cycle. Trouver les instructions exécutées simultanément ainsi que leur nombre n'est pas évident. Pour aider la lecture des résultats, l'outil essaie de sommer le nombre de cycles d'une instruction avec ceux de l'instruction précédente (jusqu'à 4 instructions). Ce calcul permet de mettre en évidence des blocs d'instructions exécutées simultanément.

3.5.3 Résultats

Les tests suivants sont réalisés sur un processeur Intel Skylake Gold capable d'exécuter 4 instructions par cycle dont au maximum deux opérations sur des nombres à virgule flottante (FLOP).

3.5.3.1 Impact des dépendances

La performance de nombreuses applications est détériorée par la présence de dépendances entre les instructions. La seule solution est alors de repenser l'algorithme pour en supprimer le plus possible. Dans cet exemple, nous montrons comment la dépendance entre instructions impacte la performance et comment utiliser l'outil `Oprofile++` pour les diagnostiquer.

L'extrait 3.17 montre le code d'un noyau de calcul généré grâce à l'outil `Kernel Generator` (voir section 3.3). Le noyau consiste en l'exécution de 8 instructions dont chacune utilise comme opérande le résultat de la précédente. En raison de ces dépendances, la performance du noyau est limitée par la latence de calcul d'une instruction (4 cycles). La performance maximale atteignable par un tel code est d'une instruction tous les 4 cycles, soit un IPC de 0.25. Ce résultat théorique est bien celui mesuré par notre outil.

```
    IPC      CYCLES      INSTS      ADDRESS     ASSEMBLY

...
  0.262        825        216       401c27     vfmadd231pd %zmm0,%zmm2,%zmm3
   0.24        800        192       401c2d     vfmadd231pd %zmm0,%zmm3,%zmm4
  0.228        886        202       401c33     vfmadd231pd %zmm0,%zmm4,%zmm5
  0.236        787        186       401c39     vfmadd231pd %zmm0,%zmm5,%zmm6
   0.25        809        202       401c3f     vfmadd231pd %zmm0,%zmm6,%zmm7
  0.271        756        205       401c45     vfmadd231pd %zmm0,%zmm7,%zmm8
  0.232        841        195       401c4b     vfmadd231pd %zmm0,%zmm8,%zmm9
   0.26        759        197       401c51     vfmadd231pd %zmm0,%zmm9,%zmm10
  0.229        808        185       401c57     vfmadd231pd %zmm0,%zmm10,%zmm11
  0.232        772        179       401c5d     vfmadd231pd %zmm0,%zmm11,%zmm12
  0.218        856        187       401c63     vfmadd231pd %zmm0,%zmm12,%zmm13
  0.244        798        195       401c69     vfmadd231pd %zmm0,%zmm13,%zmm14
  0.226        805        182       401c6f     vfmadd231pd %zmm0,%zmm14,%zmm15
  0.244        782        191       401c75     vfmadd231pd %zmm0,%zmm15,%zmm16
  0.254        784        199       401c7b     sub     $0x1,%eax
      0          0          0       401c7e     jne     4018f7 <myBench>

_7_ LOOP from 401c7e to 4018f7 size= 903
    sum(cycles)= 120060 sum(inst)= 30436 #inst= 152 IPC= 0.254 cycles/LOOP= 600
```

Extrait 3.17 – Noyau de calcul présentant une dépendance entre chaque instruction.

Ce type de dépendance est très courant dans les algorithmes d'application HPC tel que celles réalisant du calcul polynomiale. La solution pour optimiser ce type de noyau est présentée dans la section 3.3.3.3. L'optimisation est réalisée à l'aide du module d'exécution dans le désordre du processeur. Pour en profiter, il est nécessaire de donner suffisamment d'instructions à exécuter à celui-ci. Deux itérations de boucle étant indépendantes, il est possible de dérouler plusieurs itérations pour calculer plusieurs *streams* à la fois (voir explication dans la section 3.3.3.3). L'extrait 3.18 présente le profil de la performance du noyau modifié pour exécuter le calcul de 8 chaînes indépendantes. Appliquer cette optimisation à un code réel peut demander une

lourde restructuration du code mais elle est indispensable pour atteindre la performance crête de l'architecture (2 instructions flottantes par cycle).

```
------------------------------------------------------------------------------
    IPC      CYCLES      INSTS      ADDRESS     ASSEMBLY
------------------------------------------------------------------------------
...
   2.05        108        221       401c21     vfmadd231pd %zmm0,%zmm9,%zmm2
   1.81         96        174       401c27     vfmadd231pd %zmm0,%zmm10,%zmm3
    2.2        102        224       401c2d     vfmadd231pd %zmm0,%zmm11,%zmm4
   2.56         77        197       401c33     vfmadd231pd %zmm0,%zmm12,%zmm5
   2.32         97        225       401c39     vfmadd231pd %zmm0,%zmm13,%zmm6
   1.89        102        193       401c3f     vfmadd231pd %zmm0,%zmm14,%zmm7
   2.56         86        220       401c45     vfmadd231pd %zmm0,%zmm15,%zmm8
   2.05         92        189       401c4b     vfmadd231pd %zmm0,%zmm16,%zmm9
   2.06         93        192       401c51     vfmadd231pd %zmm0,%zmm2,%zmm10
   2.09         81        169       401c57     vfmadd231pd %zmm0,%zmm3,%zmm11
   1.94        108        210       401c5d     vfmadd231pd %zmm0,%zmm4,%zmm12
   1.92         97        186       401c63     vfmadd231pd %zmm0,%zmm5,%zmm13
   1.97        110        217       401c69     vfmadd231pd %zmm0,%zmm6,%zmm14
   2.61         56        146       401c6f     vfmadd231pd %zmm0,%zmm7,%zmm15
   1.73        117        202       401c75     vfmadd231pd %zmm0,%zmm8,%zmm16
   2.06         94        194       401c7b     sub     $0x1,%eax
      0          0          0       401c7e     jne     4018f7 <myBench>
------------------------------------------------------------------------------
_7_ LOOP from 401c7e to 4018f7 size= 903
    sum(cycles)= 15023 sum(inst)= 30413 #inst= 152 IPC= 2.02 cycles/LOOP= 75.1
------------------------------------------------------------------------------
```

Extrait 3.18 – L'optimisation du noyau précèdent permet de présenter au processeur 8 chaînes de calcul indépendantes.

3.5.3.2 Mesure de l'IPC

L'IPC est une métrique souvent utilisée pour décrire la performance d'une application. L'exemple précédent utilise un noyau artificiel ne comportant que des instructions de calcul. Dans cette seconde expérimentation, nous montrons que conclure de la bonne ou mauvaise performance d'un code à partir de la mesure de l'IPC n'est pas trivial. Le processeur utilisé peut théoriquement exécuter 4 instructions par cycle. Il semble donc évident d'utiliser cette valeur comme la performance maximale à atteindre par l'application. Cependant, il est important de rappeler que si les architectures Skylake peuvent exécuter 4 instructions par cycles, un maximum de deux instructions de calcul flottant peut être exécuté. Pour illustrer nos propos, la performance du noyau présenté dans l'extrait 3.19 a été mesurée grâce à notre outil `Oprofile++`.

```
CYCLES        INSTS      ADDRESS     ASSEMBLY

  128          145       4018f7     vfmadd231pd %zmm0,%zmm1,%zmm2
  132         1276       4018fd      mov     %bl,%bh
  119          305       4018ff      mov     %cl,%ch
  128          243       401901      mov     %dl,%dh
  105          229       401903     vfmadd231pd %zmm0,%zmm1,%zmm3
  109          189       401909      mov     %bl,%bh
  129         1370       40190b      mov     %cl,%ch
  142          186       40190d      mov     %dl,%dh
  119          256       40190f     vfmadd231pd %zmm0,%zmm1,%zmm4
  135          235       401915      mov     %bl,%bh
  136         1384       401917      mov     %cl,%ch
  138          231       401919      mov     %dl,%dh
  117          199       40191b     vfmadd231pd %zmm0,%zmm1,%zmm5
  117         1109       401921      mov     %bl,%bh
  137          829       401923      mov     %cl,%ch
  127          274       401925      mov     %dl,%dh
  146          303       401927     vfmadd231pd %zmm0,%zmm1,%zmm6
  128          406       40192d      mov     %bl,%bh
  127         1308       40192f      mov     %cl,%ch
  109          222       401931      mov     %dl,%dh
  138          142       401933     vfmadd231pd %zmm0,%zmm1,%zmm7
  119          303       401939      mov     %bl,%bh
  126         1409       40193b      mov     %cl,%ch
  124          209       40193d      mov     %dl,%dh
  123          242       40193f     sub     $0x1,%eax
    0            0       401942     jne     4018f7 <myBench>

_7_ LOOP from 401942 to 4018f7 size= 75
    sum(cycles)= 3158 sum(inst)= 13004 #inst= 26 IPC= 4.12 cycles/LOOP= 6.31
```

Extrait 3.19 – Noyau de calcul n'exécutant qu'une opération de calcul par cycle.

Dans cet exemple, nous mesurons l'IPC d'un noyau de calculs dont la performance est de 4.12 instructions par cycle. L'IPC mesurée est supérieur à la performance théorique du processeur car l'instruction de branchement (`jne`) de boucle n'est jamais réellement exécutée grâce au prédicteur de branchement. Contrairement à ce qu'indique cette valeur, la performance du code est très mauvaise car la micro-architecture n'exécute en fait qu'une instruction de calcul par cycle. Le processeur utilise donc la moitié de la performance disponible.

Une architecture exécute toujours un programme donné au maximum de sa capacité. Cependant, la programmation de l'algorithme ainsi que la génération du code peut être de mauvaise qualité. L'outil `Oprofile++` est donc très important pour montrer au programmeur que le code réellement exécuté peut être transformé. L'amélioration des performances d'une application est alors dépendant de la capacité du programmeur à imaginer une autre façon de réaliser ses calculs en utilisant un autre algorithme ou en réordonnant certaines instructions. Dans cet exemple

factice, nous pouvons imaginer que la réorganisation des structures de données permettent de supprimer les instructions de déplacement mémoire `mov`. Le code ainsi obtenu est présenté dans l'extrait 3.20.

```
1
2  CYCLES    INSTS     ADDRESS    ASSEMBLY
3
4     87       111      4018f7    vfmadd231pd %zmm0,%zmm1,%zmm2
5    381       457      4018fd    vfmadd231pd %zmm0,%zmm1,%zmm3
6    342       708      401903    vfmadd231pd %zmm0,%zmm1,%zmm4
7    403       860      401909    vfmadd231pd %zmm0,%zmm1,%zmm5
8    231       694      40190f    vfmadd231pd %zmm0,%zmm1,%zmm6
9    335       444      401915    vfmadd231pd %zmm0,%zmm1,%zmm7
10   236       725      40191b    sub     $0x1,%eax
11     0         0      40191e    jne     4018f7 <myBench>
12
13 _7_ LOOP from 40191e to 4018f7 size= 39
14     sum(cycles)= 2015 sum(inst)= 3999 #inst= 8 IPC= 1.99 cycles/LOOP= 4.031
15
```

Extrait 3.20 – Noyau de calcul exécutant deux opérations de calcul par cycle.

Après cette transformation, la mesure de l'IPC est alors de 2 instructions par cycle, soit deux fois moins que celui de la version précédente. Pourtant, le noyau exécute bien deux fois plus d'instructions de calcul flottant par cycle. Cette première expérimentation a pour seul objectif d'attirer l'attention de l'utilisateur sur la précaution à prendre pour conclure de la bonne ou mauvais performance d'un code en utilisant seulement la mesure de l'IPC.

3.5.4 Conclusion

Dans cette section, nous proposons un nouvel outil `Oprofile++` qui permet d'extraire les principaux noyaux de calcul d'une application et de donner à l'utilisateur son profil de performance. Contrairement à d'autres outils existants (`Vtune`, `TAU`), nous avons voulu développer un outil nécessitant le minimum de dépendances vers des bibliothèques ou des compteurs matériels. Les dépendances et la complexité sont en effet un frein à l'utilisation d'outils plus complexes. L'outil que nous développement ne dépendant que de deux compteurs présents sur la majorité des architectures et de l'outil de profilage de Linux. Ainsi, nous assurons sa compatibilité avec le plus grand nombre de plateformes. Le profil généré par `Oprofile` permet de localiser les fonctions et les modules utilisant le plus de ressources. À partir de ces informations et du profil des évènements enregistrés (nombre de cycles et nombre d'instructions), l'outil `Oprofile++` extrait les principaux noyaux de calculs sur lesquels l'analyse de performance doit se faire. Le profil généré par notre outil permet d'obtenir deux informations essentielles : le type d'instructions exécutées et leur performance. Grâce à cet outil, l'utilisateur peut estimer le gain de performance potentiel d'une optimisation du code. Cette information est essentielle, car les restructurations du code peuvent être difficiles et demander beaucoup d'efforts. Avoir une idée du gain de performance est donc primordial pour motiver la modification du code.

L'augmentation exponentielle de la complexité des architectures et la diversité des applications rend la tâche d'analyse de performance très difficile. Chaque nouvelle configuration (application / architecture) apporte de nouveaux problèmes. Le développement d'un outil réalisant le travail d'analyse de performance automatiquement est ainsi très difficile. Avec `Oprofile++`, nous avons choisi de développer un outil basique, mais qui apporte suffisamment d'informations au programmeur pour qu'il puisse, à l'aide de sa créativité, apporter les modifications nécessaires à l'amélioration du code. L'outil `Opofile++` n'a besoin que du fichier binaire pour pouvoir fonctionner bien que l'accès au code source soit recommandé pour pouvoir appliquer les transformations requises. Si cet outil s'adresse aux programmeurs particulièrement chevronnés, il n'en reste pas moins un outil essentiel et puissant lorsqu'il est bien utilisé.

Utilisation recommandée. La performance d'une application peut sensiblement varier en fonction du jeu de données utilisé. Il est important de réaliser plusieurs profils avec des jeux différents pour isoler les parties du code les plus intéressantes à optimiser. Il est souvent intéressant d'utiliser différents jeux de données ou différents drapeaux de compilation et de mesurer leur impact sur le code généré.

3.6 Conclusion

Dans ce chapitre, nous avons présenté les quatre principaux outils développés durant ce travail de thèse. Nous avons commencé par présenter les objectifs que ces outils doivent remplir et les critères de développement à respecter. Ces critères sont nécessaires pour assurer leur compatibilité avec le maximum d'architectures, leur facilité d'installation et d'utilisation. Les deux sections sont consacrées aux deux outils `DML_MEM` et `Kernel Generator` permettant la caractérisation des deux parties essentielles de la microarchitecture : le système mémoire et l'unité de calculs d'instruction à nombre flottant. Enfin, les deux dernières sections présentent les deux outils `YABM` et `Oprofile++` permettant de réaliser le suivi de performance : activité du bus mémoire ainsi que l'extraction et la caractérisation des noyaux de calculs d'une application.

Caractérisation de l'architecture.

La section 3.2 présente un premier benchmark appelé `DML_MEM`. Il permet de mesurer la performance soutenable par le système mémoire pour des accès de type *stride*. Ces accès sont très répandus dans les codes scientifiques utilisant par exemple des algorithmes RTM. Nous avons montré que la complexité des microarchitectures rend impossible la prédiction de performance et que l'utilisation d'un tel benchmark est la seule méthode pour s'assurer de la performance du système mémoire. À travers deux exemples, nous avons vu comment le code source et le compilateur utilisé pouvaient impacter la performance de codes réalisant la même tâche. Nous avons étudié l'impact de la taille des sauts (*strides*), qui pour des raisons architecturales (taille de cache, associativité) peuvent obtenir des performances très inégales. Nous avons remarqué que la totalité des coeurs n'est pas nécessaire pour saturer le bus mémoire et qu'il peut être intéressant d'en désactiver certains pour optimiser la consommation énergétique pour des codes limités par la performance mémoire. Enfin, à travers une série d'exemples, nous avons montré comment cet outil pouvait être utilisé et comment sa flexibilité de paramétrage pouvait permettre de réaliser une multitude de tests : fonctionnement des caches, taille des lignes de cache, impact des pages larges, optimisation par déroulement de boucles, performance du préchargement mémoire. Un dernier exemple nous a permis de montrer comment `DML_MEM` pouvait être utilisé pour trouver le couple optimal fréquence, nombre de coeurs, permettant de saturer le bus mémoire.

Nous avons ensuite présenté l'outil `Kernel Generator` dans la section 3.3. Ce benchmark permet de caractériser la FPU, matériel responsable de l'exécution des instructions de calculs flottants. L'outil utilise des noyaux générés en assembleur pour évaluer très précisément la performance du matériel. Le choix d'utiliser un langage bas niveau permet d'éviter toute intervention du compilateur et nous assure de réaliser des mesures très précises. Grâce aux différentes options, nous avons montré comment utiliser le générateur pour tester différents noyaux de calculs permettant de valider la performance du matériel ou d'en déceler des bogues. Le benchmark peut être utilisé pour réaliser des calculs en simple ou double précision et exécuter

des instructions vectorielles de différente taille. Nous montrons comment le `Kernel Generator` a été utilisé pour mesurer l'impact de la taille des instructions vectorielles et de l'utilisation du *turbo* sur la fréquence du processeur. Dans un second exemple, nous avons montré comment une caractéristique majeure des architectures Haswell pouvait être détectée facilement. Grâce à une option, nous avons enfin présenté comment l'outil pouvait mesurer la performance du système d'exécution dans le désordre et évaluer le nombre de *stream* de calcul indépendant qu'il parvenait à exécuter. Cette information est une caractéristique majeure pour les applications utilisant plusieurs chaînes de calcul indépendantes.

Suivi et analyse de performances.

Dans la section 3.4 nous présentons un premier outil essentiel pour l'analyse de performance. Appelé `YAMB`, il permet de réaliser le suivi de l'activité du bus mémoire en mesurant le nombre de transaction en lecture et en écriture actuellement réalisées sur le bus ainsi que le nombre de *miss* dans le dernier niveau de cache. Séparer le trafic en lecture et en écriture est une fonctionnalité essentielle de l'outil. Nous montrerons comment utiliser ce ratio dans le chapitre suivant. Pour faciliter l'analyse du code, une bibliothèque a été développée permettant depuis le code source d'annoter les parties du code étudiées. `YAMB` est basé sur `perf`, l'outil de suivi de performance de Linux. En utilisant le sous-système de performance `Perf Events`, nous assurons le maximum de compatibilité entre différentes plateformes. Enfin, nous avons étudié l'impact de l'utilisation de l'outil sur les performances de l'application étudiée. La baisse de performance mesurée sur un benchmark tel que `STREAM` est inférieur à 5%.

Enfin, dans la section 3.5, nous présentons un nouvel outil d'analyse de performance nommé `Oprofile++`. Cet outil permet d'extraire les boucles critiques d'une application (les *hot spots*) et d'extraire leur profil de performance. Grâce à cet outil, une analyse bas niveau peut être réalisée et permettre d'obtenir des pistes pour l'optimisation des codes. Dans cette section, nous discutons aussi l'utilisation de l'IPC pour caractériser la performance d'un code. Pour terminer, nous avons étudié plusieurs exemples concrets d'analyse de code et montré comment l'outil pouvait être utilisé.

Conclusion.

Dans ce chapitre, nous avons présenté quatre outils permettant de réaliser la caractérisation d'une architecture ainsi que le suivi de la performance d'une application. Ces quatre outils ont été élaborés dans le respect des critères de développement présentés dans la première section.

L'analyse de performance est un travail difficile, nécessitant de nombreuses connaissances du comportement des architectures. Pour aider le développeur dans ce travail, nous proposons quatre nouveaux outils. Chacun d'entre eux permet de répondre à un nombre limité de questions. Contrairement à des solutions existantes comme `VTune`, nous avons choisi de développer plusieurs outils indépendants répondant chacun à une question précise. Cette approche per-

met une plus grande flexibilité pour l'utilisateur. Nous avons vu à travers divers exemples, que malgré la simplicité apparente des outils, de nombreuses caractéristiques peuvent être établies. Nous espérons qu'en réduisant la complexité de l'outillage, l'adoption des outils auprès des programmeurs sera plus grande.

Les outils présentés dans ce chapitre permettent d'obtenir certaines informations sur l'architecture ou sur la performance d'une application. Cependant, leur réelle efficacité réside dans la faculté de l'utilisateur à les utiliser ensemble pour mener son travail d'analyse. Dans le chapitre suivant, nous verrons comment ces outils utilisés avec la bonne méthodologie peuvent permettre de mener une analyse très fine des codes de calculs haute performance.

Chapitre 4

Méthodologie pour l'analyse de performance et le portage de code

Des nouvelles technologies et le développement du protocole Gen-Z vont permettre l'utilisation de nouvelles architectures pour élaborer les supercalculateurs. Dans le chapitre précédent, nous avons présenté 4 outils permettant de caractériser ces architectures et d'optimiser l'exécution des applications. Dans ce chapitre, nous proposons une suite d'étapes permettant la bonne utilisation des outils.

Sommaire

4.1 Introduction . . . 186
- 4.1.1 Contributions . . . 187
- 4.1.2 Organisation du chapitre . . . 189

4.2 Étape 1 : Rechercher les dernières innovations technologiques . . . 190
- 4.2.1 Introduction . . . 190
- 4.2.2 Caractéristiques des architectures à analyser . . . 191
- 4.2.3 Application au processeur Intel Xeon 6148 . . . 193

4.3 Étape 2 : Caractériser les architectures . . . 195
- 4.3.1 Introduction . . . 195
- 4.3.2 Application au processeur Intel Xeon 6148 . . . 197

4.4 Étape 3 : Extraire les noyaux et modéliser leur performance . . . 199
- 4.4.1 Motivations et objectifs . . . 199
- 4.4.2 Identification des kernels . . . 199
- 4.4.3 Modélisation de l'équilibre des kernels . . . 200
- 4.4.4 Simple Memory Model : Modélisation de la performance mémoire . . . 201
- 4.4.5 Application des modèles Roofline et SMM . . . 201

4.5 Étape 4 : Sélectionner la plateforme adaptée . . . 203
- 4.5.1 Le coût . . . 204
- 4.5.2 La performance . . . 205

4.6 Étape 5 : Porter et optimiser le code . . . 205
- 4.6.1 Motivations . . . 205
- 4.6.2 Analyse de performance . . . 206
- 4.6.3 Application au benchmark STREAM . . . 210

4.7 Conclusion et perspectives . . . 212

4.1 Introduction

La construction d'une plateforme de calcul exascale nous oblige à relever de nombreux défis, présentés dans la section 2.2.4. Nous avons vu que les pressions énergétiques et économiques obligeaient l'industrie à repenser les architectures et les technologies utilisées pour l'élaboration des plateformes de calcul haute performance. Dans la section 2.3, nous avons listé les principales opportunités technologiques permettant de relever ces défis, parmi lesquelles :

— le développement de technologies innovantes permettant l'élaboration d'architectures (voir section 2.3.4) ;
— le développement du protocole universel `Gen-Z` (voir section 2.3.5).

Ces deux opportunités vont permettre d'augmenter le niveau d'hétérogénéité des accélérateurs utilisés dans les supercalculateurs. Dans la section 2.3.4.3, nous avons expliqué pourquoi l'utilisation d'architectures hétérogènes dans un même calculateur était indispensable pour répondre aux défis que représente l'élaboration de plateformes exascales. Depuis 2010, l'utilisation d'un accélérateur pour assister le travail des processeurs est de plus en plus répandue. Bien que cette évolution semble se confirmer, il est important de s'intéresser à la nature des accélérateurs utilisés en 2018 [1] :

— 96% des processeurs ont une architecture x86
— 91% des processeurs sont produits par le constructeur Intel.
— 92% des accélérateurs utilisés sont des GPU produits par Nvidia.

Ainsi, bien que le nombre de supercalculateurs utilisant un accélérateur ait évolué ces dernières années (28% d'entre eux en 2018), il est intéressant de remarquer que ces plateformes sont très similaires : un processeur x86 (Intel) associé à un ou plusieurs GPU (Nvidia). La forte évolution du nombre de GPU constatée ces dernières années et la prédominance de l'utilisation des processeurs graphiques (GPU) peut en partie être expliquée par leur efficacité pour exécuter les applications d'intelligence artificielle. Cependant, si leur exécution peut être réalisée par un seul type d'accélérateur, la majorité des applications de HPC sont composées de plusieurs noyaux de calcul (kernels)s pouvant avoir des besoins très différents. Pour que ces applications puissent aussi profiter de l'hétérogénéité, les plateformes doivent posséder différents types d'accélérateurs pour y exécuter les noyaux de calcul d'une application.

Cette vision que nous présentons va être facilitée par le développement du protocole universel `Gen-Z` [2] (voir section 2.3.5). Grâce à `Gen-Z` différents accélérateurs pourront facilement collaborer pour l'exécution optimale de ces applications. Avec le développement de `Gen-Z`, le déplacement de données entre accélérateurs sera facilité et la difficulté d'agrégation de différents matériels sera réduite. Nous prédisons que ce protocole, présenté dans la section 2.3.5, va révolutionner le monde de l'informatique comme peu de technologies auparavant. Entre tous les bénéfices apportés par ce protocole, la faculté de faciliter l'hétérogénéité dans les supercalculateurs est sans doute la plus importante.

1. Données calculées à partir du classement du Top500 de novembre 2018 - https://www.top500.org/lists/2018/11/
2. Gen-Z Introduction and Executive Summary - https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview-V1.pdf

4.1.1 Contributions

Les gains de performance ne viendront pas seulement de l'utilisation d'accélérateurs puissants, mais de leur diversité et de la capacité des programmeurs à bien les utiliser. Malheureusement, en l'absence de méthodes d'analyse de la performance des codes, ces architectures innovantes sont potentiellement condamnées puisque peu d'experts savent les valoriser. Ces nouvelles technologies, très différentes de celles utilisées aujourd'hui (processeur x86, GPU, DRAM), doivent être caractérisées pour prédire le gain de performance atteignable par les applications.

Pour pouvoir profiter de ces technologies et les utiliser de façon optimale, nous avons présenté dans le chapitre précédent une suite de logiciels de caractérisation et d'analyse de performance :

1. `DML_MEM` : un benchmark mémoire permettant de caractériser la hiérarchie mémoire lors de l'exécution d'applications utilisant des accès mémoires par sauts (stride) ;
2. `Kernel Generator` : un générateur de benchmarks permettant de caractériser les unités de calcul arithmétique ;
3. `YAMB` : un outil permettant de réaliser le suivi de l'activité du bus mémoire ;
4. `Oprofile++` : un l'outil d'analyse permettant d'extraire et de caractériser les zones de codes responsables de la majorité du temps d'exécution de l'application.

Dans ce chapitre, nous présentons une méthodologie en 5 étapes qui permet aux utilisateurs de modéliser les performances de leur code, de les projeter sur de nouvelles architectures et de les optimiser (voir figure 4.1). L'objectif de ce chapitre est de présenter une méthodologie adaptée, permettant d'utiliser efficacement les outils développés durant ce travail de thèse afin de réaliser la caractérisation des architectures ainsi que le portage des applications sur ces nouveaux accélérateurs.

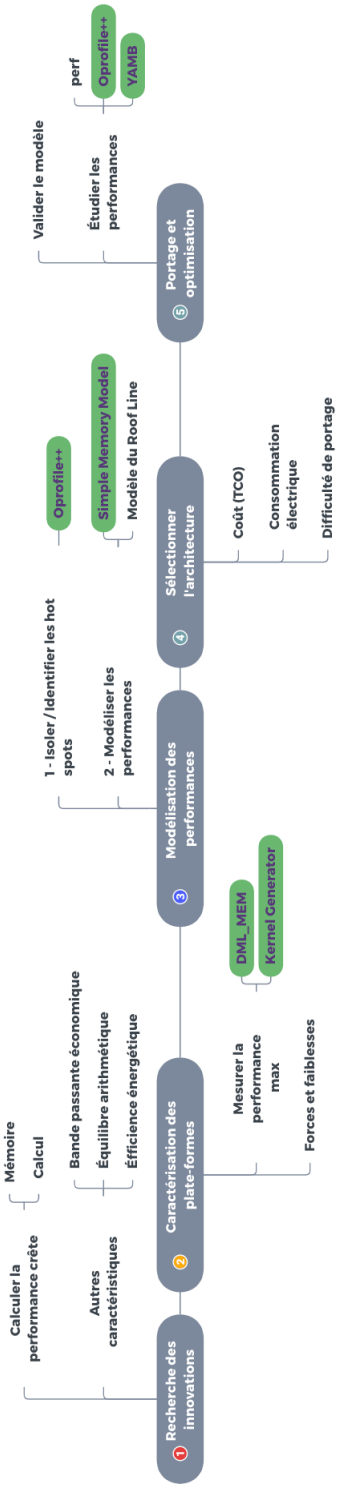


FIGURE 4.1 – Méthodologie en 5 étapes pour caractériser et optimiser une application sur une nouvelle architecture.

Délimitation de l'analyse L'analyse et l'optimisation des performances d'une application peuvent être réalisées à plusieurs niveaux : à l'échelle d'une grappe de serveurs, du processeur, d'un coeur, etc. Pour que le travail soit réalisable durant la thèse, nous avons délimité un certain cadre pour appliquer notre méthodologie (voir figure 4.2). Le premier cadre consiste à analyser les applications qui sont exécutées sur une plateforme homogène, c'est-à-dire que l'application est exécutée sur plusieurs serveurs avec la même configuration (matérielle et logicielle). Lorsque la programmation à mémoire distribuée a bien été réalisée, l'exécution sur les différents serveurs devrait être similaire. Notre analyse de performance peut alors se poursuivre sur un seul serveur. Les problèmes de performance liés à l'exécution sur plusieurs serveurs ne sont pas abordés dans cette analyse, mais peuvent l'être grâce à l'utilisation d'outils tels que `Extrae` [Rod], `Paraver` [Pil+95] ou `Tau` [SM06]. Notre approche s'intéresse aux applications HPC, dont la majorité de l'exécution est passée à exécuter des instructions de calcul. Les applications dont la performance est limitée par celle du système de stockage, du réseau ou du système d'exploitation ne sont pas la priorité de la méthodologie présentée. Cependant, avec une certaine expérience ces limitations peuvent être identifiées par nos outils. Les applications que nous ciblons dans notre analyse sont des codes dont la majorité du temps d'exécution se déroule seulement dans quelques pourcentages des lignes de codes. Nous appelons ces zones des points chauds, points chauds (hot spots) ou encore kernels. Une application possédant des hot spots est le gage d'un potentiel d'amélioration des performances. Notre approche s'intéresse particulièrement aux applications dont la performance est limitée par la performance du système mémoire (memory bound) ou celle des unités de calcul (compute bound).

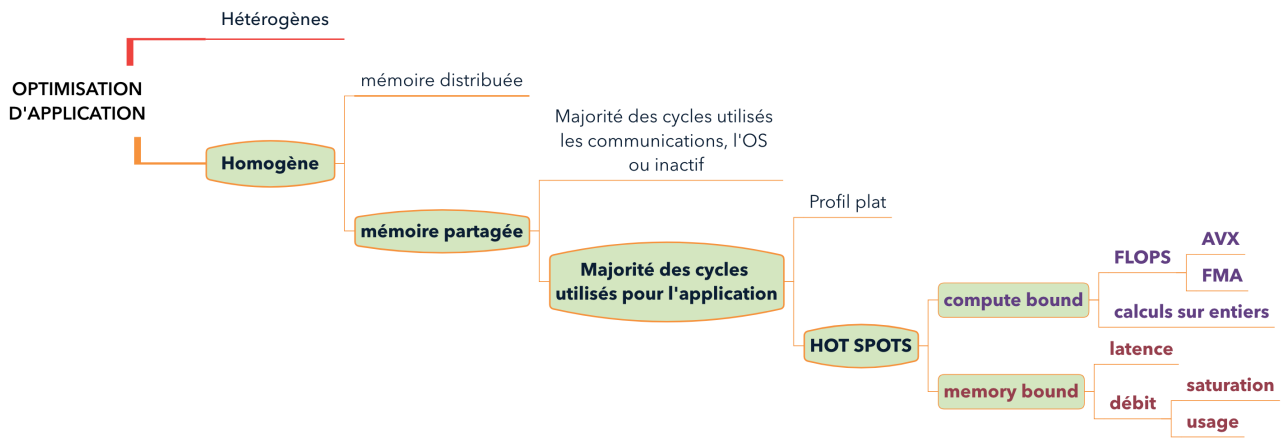


FIGURE 4.2 – Délimitation de l'analyse proposée.

4.1.2 Organisation du chapitre

Ce chapitre présente les cinq étapes de la méthodologie et suit la structure suivante :

— La section 4.2 présente la première étape qui consiste à se tenir au courant (veille technologique) de toutes les nouveautés technologiques ayant un potentiel pour être utilisées

dans les centres de calcul. Nous discutons des caractéristiques clefs qu'il est nécessaire de quantifier pour estimer le potentiel de chaque architecture.

— La section 4.3 discute des méthodes pour calculer ou mesurer la performance d'une architecture. Nous montrons comment deux des outils présentés dans le chapitre précédent peuvent être utilisés pour cela.

— La section 4.4 s'intéresse aux applications et notamment à la modélisation de leur performance. Pour ce faire, nous présentons différents outils permettant d'isoler les hot spots d'une application ainsi qu'un modèle de performance basé sur la performance du système mémoire.

— La section 4.5 discute ensuite des principaux facteurs à étudier pour réaliser le choix des architectures à utiliser pour l'élaboration d'une nouvelle plateforme de calcul.

— La section 4.6 s'intéresse enfin au portage du code sur les plateformes sélectionnées précédemment. Nous y présentons une suite d'étapes à suivre pour valider la bonne performance des noyaux et, dans le cas contraire, les étapes à suivre pour optimiser leur performance.

Pour illustrer les différentes étapes, la méthodologie est appliquée à l'étude des performances de la fonction *triadd* (voir extrait 4.1) du benchmark `STREAM` [McC95]. Cette expérimentation est réalisée sur un processeur Intel *Xeon Gold 6148* possédant 20 coeurs. Les matrices utilisées mesurent chacune 19.6 GB.

```
1 for (j=0; j < STREAM_ARRAY_SIZE; j++)
2     A[j] = B[j] + scalar * C[j];
```

Extrait 4.1 – Fonction *triad* extraite du benchmark `STREAM` [McC95].

4.2 Étape 1 : Rechercher les dernières innovations technologiques

4.2.1 Introduction

4.2.1.1 Motivations

Le domaine du calcul haute performance est très concurrentiel. Les industries ayant recours à ces plateformes doivent être à la pointe de la technologie au risque de se faire dépasser par un concurrent. Comme vu dans la section 2.3.4.3, la grande majorité des plateformes utilisent les mêmes architectures. Jusqu'à aujourd'hui, peu d'entreprises se sont démarquées par l'utilisation d'un nouvel accélérateur.

Aujourd'hui, les choix d'architectures disponibles se limitent à quelques architectures : les processeurs (Intel, AMD, IBM) et les GPU (NVIDIA, AMD, Xeon Phi). Les technologies émergentes présentées dans la section 2.3, vont permettre le développement de nouvelles architectures. Grâce au protocole `Gen-Z`, il sera plus facile de les utiliser ensemble au sein d'une même

plateforme. Il est donc indispensable pour les industries, mais aussi pour les constructeurs tels que `Hewlett Packard Enterprise`, de connaître et d'utiliser les meilleures d'entre elles. Porter son application sur une nouvelle plateforme nécessite d'investir du temps et de l'argent. Cette décision très importante se complexifie avec l'augmentation du nombre d'architectures différentes.

4.2.1.2 Objectifs

Le premier travail des développeurs et des architectes de plateformes Calcul Haute Performance (HPC) est d'être en constante recherche des dernières innovations technologiques. Cela peut être l'annonce d'un nouveau processeur, d'une nouvelle technologie mémoire ou bien de nouveaux algorithmes ou de nouvelles optimisations. Il est très important de se tenir à l'état de l'art ou même en avance pour anticiper les nouveautés.

Le but principal de cette étape est de répertorier toutes les plateformes et technologies potentiellement intéressantes pour le calcul haute performance. Certaines caractéristiques clés sont calculées à partir des spécificités techniques des architectures, d'autres sont mesurées dans l'étape suivante lorsque l'accès aux plateformes est disponible.

Dans notre vision, la majorité des lignes de codes continueront d'être exécutées sur des architectures semblables à celles d'aujourd'hui (x86 et PowerPC). Seuls les kernels seront déportés sur les accélérateurs adéquats. Il est donc nécessaire de continuer à s'y intéresser et à les caractériser. Les processeurs présents dans ces nouvelles générations de plateformes pourraient alors être bien différents de ceux utilisés actuellement. En effet, si les kernels des applications ne sont plus exécutés sur ces processeurs, les caractéristiques recherchées seront différentes. Les accélérateurs actuels continueront d'avoir leur rôle à jouer. Les GPU se montrent extrêmement efficaces pour les algorithmes d'apprentissage par machine et d'intelligence artificielle. L'objectif de l'industrie est de trouver des accélérateurs aussi efficaces pour l'exécution d'autres types d'algorithmes.

4.2.2 Caractéristiques des architectures à analyser

Pour la suite de l'analyse, il est nécessaire de récupérer des caractéristiques clefs pour chaque architecture. La majorité des applications ont besoin d'un bus mémoire très performant. Cependant, certaines parties du code, séquentielles ou utilisant seulement les unités arithmétiques et logiques, auront d'autres besoins et devront être portées sur des architectures différentes. Il ne faut donc négliger aucune architecture qui pourrait s'avérer intéressante pour une partie du code ou une autre application. Nous avons regroupé cinq caractéristiques clés qui sont importantes à obtenir pour évaluer le potentiel d'une architecture :

1. La bande passante mémoire mesurée en GB/s ;
2. La puissance de calcul mesurée en $FLOPS$;
3. La bande passante économique mesurée $GB/seconde/dollar$;
4. L'équilibre arithmétique mesuré en $flops/GB/s$;

5. L'efficience énergétique mesurée $flop/seconde/watt$.

La bande passante mémoire notée $\texttt{MEMORY}_{\texttt{peak}}$ et mesurée en GB/s, nécessite de connaître plusieurs caractéristiques du système mémoire pour être calculée. Il existe différentes technologies mémoires permettant d'écrire entre une, deux ou quatre fois par cycle sur chaque ligne du bus. On parle alors de mémoire Single Data Rate (SDR), Double Data Rate (DDR) et Quad Data Rate (QDR). La fréquence seule ne permet donc pas d'indiquer combien de transferts peuvent être réalisés par seconde, il faut aussi connaître le débit de données. Pour éviter les confusions, on parle alors de *Mega Transferts* par seconde (MT/s) noté `MTS`. La fréquence et le MTS des mémoires sont deux grandeurs différentes qui sont souvent confondues, par les constructeurs eux-mêmes. Par exemple la DDR4-2666, signifie que la RAM a une fréquence de 1333 MHz. La DDR4 étant une mémoire DDR, une mémoire DDR4-2666 aura un débit de 2666 `MTS`. Pour calculer le débit mémoire disponible, il faut ensuite connaître le nombre de lignes reliant la mémoire au processeur, que l'on note bus_width. Les architectures x86 récentes utilisent des canaux (ou *memory channels*) de 64 bits. Pour obtenir une grande bande passante mémoire, les architectures utilisent plusieurs canaux mémoire notés $nb\_channels$. Ainsi, la bande passante maximum théorique $\texttt{MEMORY}_{\texttt{peak}}$ peut alors être calculée avec la formule suivante :

$$MEMORY_{peak} = MTS \times bus\_width \times nb\_channels \tag{4.1}$$

La puissance de calcul ou la performance crête de calcul, est mesurée en nombre d'opérations à virgule flottante par seconde (FLOPS) et notée $\texttt{FLOPS}_{\texttt{peak}}$. Pour la calculer, nous adaptons la notation proposée dans de précédents travaux [Dol15].

Pour calculer la performance maximale théorique d'un processeur, nous commençons par calculer le nombre maximal d'opérations à virgule flottante (FLOP) exécutables par cycle. Cette performance est notée $\texttt{FLOP}_{\texttt{cycle}}$ et mesurée en `flop/cycle`. Pour la calculer, plusieurs données techniques sont nécessaires. Tout d'abord, il est nécessaire de connaître la taille des instructions vectorielles. La taille de ces instructions (SIMD) et leur disponibilité dépend de l'architecture. Elle est mesurée en `flop/operation`. Ensuite, il faut connaître le nombre maximal d'opérations exécutées par opération, mesuré en `operation/instruction`. Sur les processeurs modernes, ce sont les instructions Fused Multiply Add (une multiplication et une addition fusionnées (FMA)) qui sont capables d'exécuter deux opérations en un cycle. Enfin, les processeurs étant généralement superscalaires (voir Annexe A.2.4.2), il faut obtenir le nombre d'instructions pouvant être exécutées en un cycle. Cette valeur est mesurée en `instructions/cycle` et peut être trouvé à l'aide de la documentation des unité de calcul à virgule flottante (FPU) (voir section A.2.3.4). À l'aide de ces trois caractéristiques techniques, $\texttt{FLOP}_{\texttt{cycle}}$ peut être calculé grâce à la formule suivante :

$$FLOP_{cycle} = \frac{flop}{operation} \times \frac{operations}{instruction} \times \frac{instructions}{cycle} \tag{4.2}$$

Une fois la performance maximale de la microarchitecture obtenue, il faut calculer la per-

formance crête théorique atteignable par le processeur, notée $\texttt{FLOPS}_{\texttt{peak}}$ mesurée en FLOPS. Pour cela, il faut connaître la fréquence atteignable par le processeur lors de l'exécution des instructions SIMD utilisées pour calculer $\texttt{FLOP}_{\texttt{cycle}}$. Pour éviter des problèmes de surchauffe, le processeur doit abaisser sa fréquence lorsqu'il utilise de telles instructions. Un tableau de correspondance entre le type d'instructions utilisées et la fréquence soutenable est généralement donné par le constructeur. Enfin, il faut avoir le nombre de coeurs disponibles sur le processeur. La performance maximale théorique $\texttt{FLOPS}_{\texttt{peak}}$ peut alors être calculée à l'aide de la formule suivante :

$$FLOPS_{peak} = FLOP_{cycle} \times \frac{cycle}{seconde} \times nombre\ de\ coeurs \tag{4.3}$$

La bande passante économique mesurée en $GB/seconde/dollar$ représente le débit de données transférables par seconde pour le prix de la plateforme. Deux facteurs importants dans le choix de la plateforme entrent ici en jeu : la bande passante disponible, facteur limitant pour la majorité des codes, ainsi que l'économie qui est souvent l'élément de décision ultime. On cherchera les plateformes avec la plus grande bande passante économique.

L'équilibre arithmétique mesuré en $flops/GB/s$ représente le nombre d'opérations réalisables pour chaque donnée transférée depuis la mémoire. Cette valeur permet d'estimer l'équilibre entre le calcul et le débit mémoire d'une plateforme. Une grande valeur signifiera que la plateforme est plutôt destinée à des codes intensifs en calcul. À l'inverse, une valeur faible signifiera que la plateforme est adaptée à des codes nécessitant beaucoup d'accès mémoire. Pour la majorité des applications, on cherchera à obtenir une valeur petite.

L'efficience énergétique mesurée $flop/seconde/watt$ représente le rapport d'opérations flottantes par watt d'énergie consommée. Comme discuté dans la section 2.2.4.2, la consommation électrique du supercalculateur est une contrainte majeure pour le projet Exascale. Il est donc important de privilégier des architectures avec les meilleurs rendements énergétiques. On cherche ici à obtenir la plus grande valeur possible.

Le calcul des caractéristiques par les données techniques des architectures a l'avantage de permettre d'évaluer rapidement leur potentiel sans y avoir accès. Dans la suite de cette partie, nous présentons comment certaines caractéristiques, comme la bande passante ou la puissance crête d'un processeur, peuvent être calculées.

4.2.3 Application au processeur Intel Xeon 6148

Pour illustrer la présentation de la méthodologie, nous utilisons l'exemple d'un processeur Intel Xeon Skylake 6148 possédant 20 coeurs et une fréquence de base de 2,4 GHz. Une configuration à deux processeurs est présentée sur la figure 4.3. Ce modèle de processeur possède les caractéristiques suivantes :

— **Mémoire :** le processeur étudié possède 6 canaux mémoire le connectant à 6 barrettes mémoires cadencées à 2666 MT/s.

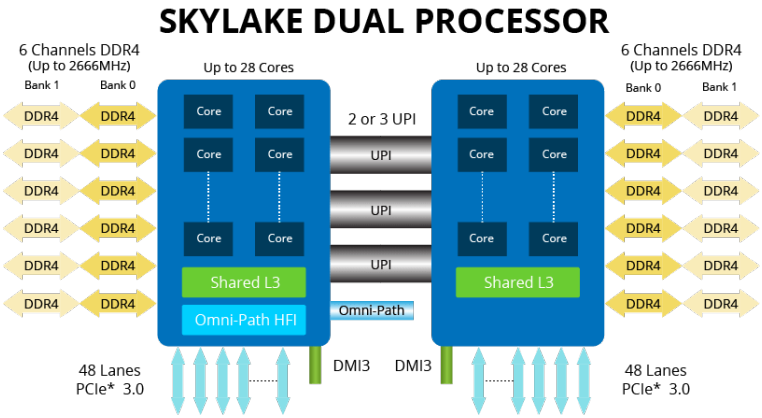


FIGURE 4.3 – Architecture d'une plateforme avec deux processeurs Xeon Skylake (source [asp]). Chaque processeur est relié à la mémoire par 6 canaux mémoire cadencés, dans notre cas, à 2666 MTS.

— **ALU :** les processeurs de la gamme Xeon Gold 6 possèdent tous deux unités AVX-512 capables d'exécuter chacune 2 instructions vectorielles de 512 bits (AVX-512), dont FMA.
— **Fréquence :** pour une même architecture (ici Intel Skylake), chaque modèle de processeur a ses propres plages de fréquences utilisables qui peuvent être consultées en ligne [wik]. La fréquence utilisable dépend essentiellement de la consommation électrique du processeur et de sa température (dépendant de la qualité du système de refroidissement). Ainsi, les fréquences soutenables par le processeur dépendent du nombre de coeurs utilisés, de la taille des instructions exécutées (normal, AVX-2 ou AVX-512) et de la disponibilité du Turbo. Pour l'exécution d'instruction SISD (Single Instruction Single Data) avec le turbo actif sur les 20 coeurs, la fréquence maximale atteignable est de 3,1 GHz. En fonction de l'utilisation du turbo et de la qualité du système de refroidissement, le processeur Skylake 6148 peut utiliser des fréquences allant de 1,6 GHz à 2,2 GHz [wik].

4.2.3.1 Performances mémoires théoriques

Le processeur Xeon Skylake 6148 possède 6 canaux mémoire pour accéder, dans notre expérimentation, à une mémoire DDR4-2666. En appliquant l'équation 4.1 nous obtenons une bande passante maximale de : $2666 \times 8 \times 6 = 128\ GB/s$. À cause de la loi de Little [LG08], le processeur doit être capable de lancer plusieurs transactions simultanément (*outstanding load*) pour saturer le bus mémoire et atteindre cette performance. Nous avons montré dans la section 3.2.3.4 que ce processeur nécessitait d'avoir au moins 15 coeurs actifs pour y parvenir.

4.2.3.2 Puissance de calcul théorique

Le processeur étudié est un processeur superscalaire capable d'exécuter jusqu'à 4 instructions par cycle, dont deux opérations à virgule flottante. Ces opérations pouvant être des

instructions FMA vectorielles de 512 bits. Il est donc possible de calculer sur chaque ALU, une multiplication et une addition par cycle sur 8 éléments simultanément. On peut ainsi calculer la performance crête de ce processeur en appliquant l'équation 4.3. Suivant la fréquence utilisable par le processeur (dépendant de la température) la performance crête théorique ($\texttt{FLOPS}_{\texttt{peak}}$) est comprise entre $8 \times 2 \times 2 \times 1.6 \times 20 = 1024$ GFLOPS et $8 \times 2 \times 2 \times 2.2 \times 20 = 1408$ GFLOPS.

Cependant, pour comparer la performance de l'application avec ce résultat, il faut que la nature du code puisse utiliser des instructions FMA vectorisées. Il peut être intéressant de disposer d'une fourchette de performance lorsque la totalité du parallélisme est utilisée ou non. Quand le processeur n'utilise pas d'instruction AVX-512, le processeur est capable d'atteindre 3.1 GHz lorsque les 20 coeurs sont actifs. En reprenant l'équation 4.3, la performance optimale d'une telle application serait : $\texttt{FLOPS}_{\texttt{max}} = 1 \times 1 \times 2 \times 3.1 \times 20 = 124$ GFLOPS.

4.2.3.3 Équilibre arithmétique

L'équilibre arithmétique du processeur permet d'évaluer s'il est approprié pour un code nécessitant une grande bande passante ou plutôt de bonnes performances de calculs. En réutilisant les deux caractéristiques précédemment calculées, on peut calculer $\text{EQUILIBRE}_{non\_avx}$ et $\text{EQUILIBRE}_{avx\_512}$ qui bornent la performance inférieure et supérieure de ce processeur. On obtient ainsi $\text{EQUILIBRE}_{non\_avx} = \frac{124}{128} = 0.97\ flop/byte$ et $EQUILIBRE_{avx\_512} = \frac{1408}{128} = 11$ flopbyte. L'équilibre arithmétique est utilisé pour construire le modèle du Roofline présenté dans la section 2.4.2.5.

4.3 Étape 2 : Caractériser les architectures

Lors de la première étape, les caractéristiques des architectures ont été rassemblées pour sélectionner celles avec le meilleur potentiel. Pour l'étape deux, il est nécessaire d'avoir accès aux différentes architectures pour réaliser une caractérisation fine de leur comportement et de leurs performances.

4.3.1 Introduction

Pour pouvoir estimer la bonne ou mauvaise performance d'un code sur une plateforme, il est nécessaire d'avoir une performance de référence avec laquelle la comparer. Grâce à cette référence, il est ensuite possible d'estimer les gains de performance dont pourrait profiter une application suite à son optimisation ou à son portage sur une nouvelle architecture. Un modèle largement utilisé dans le domaine de l'analyse de performance est celui du Roofline, présenté dans la section 2.4.2.5. Pour sa construction, il faut disposer de deux caractéristiques de l'architecture : le débit mémoire (mesuré en GB/s) et le débit de calcul (mesuré en FLOP). Pour les obtenir, deux méthodes sont possibles (voir figure 4.4). La première est de les calculer à partir des données techniques de l'architecture et la deuxième est de la mesurer.

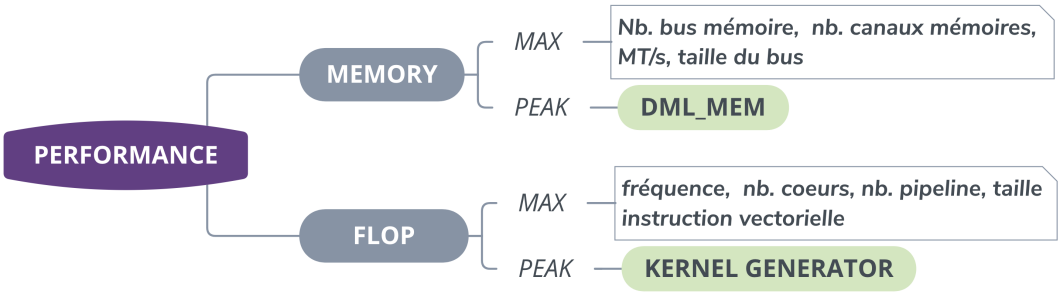


FIGURE 4.4 – Les performances d'une architecture peuvent être calculées à l'aide des caractéristiques techniques ($\texttt{FLOPS}_{\texttt{peak}}$, $\texttt{MEMORY}_{\texttt{peak}}$) ou mesurées à l'aide d'applications spécialisées ($\texttt{FLOPS}_{\texttt{max}}$, $\texttt{MEMORY}_{\texttt{max}}$).

4.3.1.1 Performance maximale théorique

L'implémentation *naïve* [Wil08] du modèle du Roofline utilise les performances théoriques de la microarchitecture. Celle du système mémoire est notée ($\texttt{MEMORY}_{\texttt{peak}}$) et celle du processeur ($\texttt{FLOPS}_{\texttt{peak}}$). Malheureusement, à cause de la complexité des architectures, il est très rare qu'une application atteigne une performance égale à la performance théorique. Même des applications spécialisées telles que `STREAM` [McC95] et `HPL` [DLP03] n'y parviennent pas. Il est donc très rare de voir des applications industrielles atteindre la performance maximale théorique des processeurs.

Lors de l'étape 1, nous avons utilisé les caractéristiques techniques des architectures pour réaliser un premier tri, sans même y avoir accès physiquement. L'objectif de cette deuxième étape est d'obtenir des valeurs de références pour pouvoir projeter et apprécier les performances d'une application.

4.3.1.2 Performance maximale mesurée

Afin de connaître précisément les performances d'une architecture, il est conseillé d'utiliser des applications spécialisées pour chaque caractéristique. Une application industrielle peut être difficile à porter sur une nouvelle architecture. Il est donc peu recommandé de réaliser ce portage dans le simple but de caractériser cette dernière. Il est préférable d'utiliser des benchmarks, plus courts et plus facilement portables. Après le portage de l'application sur la plateforme choisie, il sera possible de comparer la performance mesurée de l'application (performance *maximale*[3]), avec la performance théorique atteignable (performance *crête*[3]).

Pour obtenir les deux caractéristiques qui nous intéressent ici (notées $\texttt{MEMORY}_{\texttt{max}}$ et $\texttt{FLOPS}_{\texttt{max}}$), il est courant d'utiliser des benchmarks de références. Par exemple pour mesurer la bande passante mémoire maximale atteignable, on peut utiliser le benchmark *Stream* [McC95]. Afin d'obtenir les latences des différents niveaux de caches, *lmbench* [SI02] peut alors être utilisé. Pour mesurer le nombre maximum d'opérations sur un nombre flottant exécutable par seconde, on peut utiliser le benchmark HPL.

L'avantage de cette approche est sa facilité. Il suffit de compiler le benchmark voulu et de

l'exécuter pour obtenir les résultats. L'inconvénient est que la mesure est dépendante de la qualité du code et du compilateur. Il peut arriver qu'en voulant mesurer spécifiquement un composant, les performances soient dégradées par une autre partie de la microarchitecture, par exemple, si l'on cherche à mesurer la bande passante maximale atteignable par un seul coeur avec un code simple lisant un tableau de données. Sur un processeur récent tel qu'un Xeon Skylake, on s'attendrait à obtenir une valeur proche du maximum théorique de 128 GB/s, calculé dans lors de l'étape précédente. Cependant, à cause de la Loi de Little [LG08] et de la taille de la queue de chargement (*outstanding load queue*), il faut plus de quinze coeurs actifs pour saturer la bande passante [Joh10]. Il faut donc une certaine expérience des outils et des microarchitectures pour apprécier les résultats mesurés. Il peut ainsi être nécessaire d'avoir plusieurs codes de benchmark à exécuter pour valider de différentes façons le comportement de la microarchitecture.

4.3.2 Application au processeur Intel Xeon 6148

Nous présentons ici comment les outils développés durant la thèse permettent de caractériser la microarchitecture du processeur Intel Xeon Skylake 6148. Nous mesurons le débit mémoire maximale du bus mémoire ($\texttt{MEMORY}_{\texttt{max}}$) à l'aide de `DML_MEM` (voir section 3.4) et la puissance de calcul du processeur ($\texttt{FLOPS}_{\texttt{max}}$) grâce au `Kernel Generator` (voir section 3.3).

4.3.2.1 Débit mémoire maximal

Pour caractériser le système mémoire du processeur, nous paramétrons l'outil `DML_MEM` pour réaliser des accès mémoire avec des sauts en mémoire de la taille d'une ligne de cache. Pour nous assurer de la capacité du processeur à générer suffisamment de transactions mémoires pour saturer le bus (voir Loi de Little section A.3.2.4), nous exécutons le benchmark sur les 20 coeurs disponibles. La commande résultante pour cette expérimentation est la suivante :

```
mpirun -np 20 numactl dml_mem  --steplog 0 --unroll 8
                               --type read --stride 64 --matrixsize 5000
```

Lors de l'étape 1, nous avions calculé une performance crête théorique MEMORY_{peak} de 128 GB/s. A l'aide de `DML_MEM`, nous obtenons une bande passante mémoire $\texttt{MEMORY}_{\texttt{max}}$ de 105 GB/s.

4.3.2.2 Performance de calcul

Pour mesurer la performance maximale atteignable ($\texttt{FLOPS}_{\texttt{max}}$) par le processeur, nous utilisons le générateur de benchmark `Kernel Generator`. Nous avons paramétré le générateur pour évaluer le nombre maximum d'opérations FMA AVX-512 réalisable sur un nombre flottant à double précision. Pour cela, nous avons utilisé la commande présentée ci-dessous. Nous générons un kernel de 14 instructions FMA pour réduire le coût de gestion de la boucle (incrémentation et comparaison) et masquer la latence des instructions.

3. Le classement du Top500 utilise les notations R_{max} et R_{peak} pour désigner la performance maximale atteignable et la performance crête (*peak*) théorique d'un supercalculateur.

```
./kg -W 512 -O ffffffffffffff -P double -S 100 -L 120000000
```

Le benchmark généré ne possède pas de version multicoeur, nous lançons donc indépendamment 20 exécutions du binaire qui sont accrochées à un coeur différent grâce à un paramètre du benchmark. Pour cette expérience, nous avons commencé par désactiver le turbo du processeur, et limité sa fréquence à 1,6 GHz. La performance mesurée est de 998,57 GFLOPS. Ensuite nous avons activé le turbo, et laissé le processeur choisir lui même sa fréquence. L'extrait 4.2 montre les résultats donnés par le benchmark pour l'exécution sur un des 20 coeurs. Le benchmark mesure sa fréquence effective et trouve effectivement la valeur de 2,2 GHz renseignée par Intel dans sa documentation. La performance maximale d'un coeur mesurée est de 69,2 GFLOPS, approchant le maximum théorique de 70 GFLOPS calculé dans l'étape 1 (voir section 4.2.3.2).

```
------------------- INSTRUCTIONS SUMMARY -------------------------------
_label_|   NB INSTRUCTIONS      Time    FREQUENCY    inst/sec       IPC
_value_|      168000000000      38.8          2.2     4.33e+9      2.01
------------------------ FLOP SUMMARY ------------------------------------
 PRECISION     FLOP/cycle          FLOP/second
    Single              0                    0
    Double           32.0             6.92e+10
------------------------------------------------------------------------
```

Extrait 4.2 – Résultat de l'exécution du benchmark sur un coeur avec le turbo activé

Le benchmark du `Kernel Generator`, ne possède pas de version multicoeur et ne présente que les résultats de la performance d'un coeur. Pour nous assurer de l'exécution du code sur les différents coeurs, nous avons utilisé un outil développé chez HPE appelé *mygflops.sh*. Il permet de compter les instructions flottantes simples et doubles précisions exécutées sur un processeur. Le résultat est présenté dans l'extrait 4.3. La performance maximale mesurée du processeur $\texttt{FLOPS}_{\texttt{max}}$ est de 1372.78 GFLOPS, proche du maximum théorique FLOPS_{peak} de 1408 GFLOPS, calculé lors de l'étape 1 (voir section 4.2.3.2).

```
Single-precision SSE/AVX :        0.00 GFlop/s  --    0.0% of Flops
Double-precision SSE/AVX :     1372.78 GFlop/s  -- 100.0% of Flops
   0.0% scalar  64-bit SSE/AVX instructions (  0.0% of fp instructions)
   0.0% packed 128-bit SSE/AVX instructions (  0.0% of fp instructions)
   0.0% packed 256-bit AVX instructions     (  0.0% of fp instructions)
 100.0% packed 512-bit AVX instructions     (100.0% of fp instructions)
```

Extrait 4.3 – Résultat de l'outil *myflops.sh* utilisé pour compter les instructions flottantes exécutées sur un processeur.

4.4 Étape 3 : Extraire les noyaux et modéliser leur performance

Les deux premières étapes de la méthodologie s'intéressent à la recherche et à la caractérisation de nouvelles plateformes. La troisième étape concerne la modélisation des performances de l'application. Celle-ci est indépendante des deux premières étapes, car elle ne dépend pas de l'architecture envisagée pour porter l'application.

4.4.1 Motivations et objectifs

En introduction de cette partie, nous avons rappelé la définition des hot spots. Ceux-ci sont particulièrement présents dans les applications de calcul haute performance et correspondent généralement aux kernels de calcul. Ces zones de codes possèdent un fort potentiel pour l'amélioration de la performance de l'application. Si une application passe 99% de ses cycles dans l'exécution d'une fonction, une amélioration d'un facteur 10 de celle-ci entraînera une amélioration du même facteur de l'application. Le travail du programmeur est donc d'identifier et d'accélérer ces parties en priorité.

Les applications réelles utilisées en production dépassent souvent les dizaines de milliers de lignes de codes. Porter et optimiser la totalité d'une application serait complexe et contre-productif. De plus, la performance de chaque kernel peut être limitée par des parties différentes du matériel (memory bound, compute bound, etc), il est donc nécessaire de les porter individuellement sur différentes plateformes.

L'objectif de cette étape est d'identifier ces zones clés du code et de modéliser leur performance en fonction des performances de la bande passante mémoire (GB/s) ou du processeur (FLOPS) en calculant leur intensité opérationnelle notée $\mathtt{OI}_{\mathtt{kernel}}$. La majorité des codes étant limité par le débit mémoire, nous présentons un modèle basé sur la performance du bus mémoire.

4.4.2 Identification des kernels

De nombreux travaux sont réalisés pour identifier et extraire les kernels d'une application [Cas+15 ; BW13]. L'outil de profilage `perf` [De 10] permet d'extraire un sommaire de l'exécution d'une application en représentant son arbre d'appel. L'extrait 4.4 présente le résultat donné par la commande `perf` lors de l'exécution du benchmark `STREAM`. Cette commande permet de rapidement identifier les kernels et leur part de responsabilité dans le temps d'exécution de l'application. Elle nécessite d'exécuter l'application complète. Cependant, cet exercice ne doit être réalisé qu'une seule fois en début d'analyse pour identifier les zones de code sur lesquelles l'analyse doit se poursuivre.

```
# Samples: 116K of event 'cycles:ppp'
# Event count (approx.): 2744862582690
#
# Children      Self  Command          Shared Object       Symbol
# ........  ........  ...............  ..................  .......
    99.52%     0.00%  Stream.SKL.128   [unknown]           [k] 0000000000000000
             |
              --99.52%--0
                        |
                         --99.16%--0xadf96
                                   |
                                   |--25.35%--tuned_STREAM_Add
                                   |--25.34%--tuned_STREAM_Triad
                                   |--20.36%--tuned_STREAM_Copy
                                   |--17.27%--tuned_STREAM_Scale
                                    --10.74%--main
```

Extrait 4.4 – Exemple d'utilisation de perf avec la commande `perf record -g -F 97` lors de l'exécution du benchmark `STREAM`. Le rapport d'exécution est obtenu avec la commande `perf report -stdio`.

4.4.3 Modélisation de l'équilibre des kernels

Chaque kernel peut être porté sur un accélérateur différent. Ainsi, ils doivent être analysés indépendamment les uns des autres. L'objectif de la modélisation est de comprendre les performances de l'application, et d'identifier si elles sont limitées par les performances du système mémoire (memory bound) ou par la capacité de calcul du processeur (compute bound). La modélisation des performances permet ensuite de réduire le nombre d'architectures envisagées pour le portage de l'application. Cette étape permet d'éviter d'investir du temps et de l'argent dans des solutions inefficaces pour l'application étudiée.

Pour connaître lequel du système mémoire ou du processeur est le goulot d'étranglement (bottleneck), le `Roofline Model` (voir section 2.4.2.5) peut être utilisé. Nous conseillons de construire le modèle à l'aide des caractéristiques mesurées lors de l'étape 2 ($\texttt{FLOPS}_{\texttt{peak}}$ et $\texttt{MEMORY}_{\texttt{peak}}$) pour avoir une meilleure estimation des performances réellement atteignables. Pour établir le modèle d'un kernel, il est nécessaire de calculer l'intensité opérationnelle notée $\texttt{OI}_{\texttt{kernel}}$ et mesurée en $flop/byte$. Elle représente le nombre d'opérations réalisables par le processeur pour chaque donnée transférée depuis la mémoire. Ce calcul se fait à partir de la lecture du code source, ce qui motive le besoin d'identifier individuellement les kernels. L'utilisation du modèle permet ensuite de visualiser les kernels ayant le plus grand potentiel d'amélioration de performance.

4.4.4 Simple Memory Model : Modélisation de la performance mémoire

La majorité des codes HPC exécutée sur des architectures modernes voient leurs performances limitées par celle de la bande passante mémoire. Nous avons développé un modèle de performance simple, permettant de modéliser et valider les performances d'un code facilement. Pour réaliser cette modélisation, le développeur doit avoir accès au code source de l'application à porter. Pour un kernel donné, il faut compter le nombre d'accès mémoire en distinguant les accès en lecture et ceux en écriture. Il est important de distinguer les accès en lecture et en écriture, car nous utiliserons leur ratio pour valider le bon comportement de la microarchitecture avec l'outil `YAMB` (voir section 3.4). En effet, nous montrons dans notre expérience que la saturation du bus mémoire n'est pas un indicateur suffisant pour conclure de l'efficacité ou non d'un code.

Cette modélisation est faisable seulement si les kernels du code ont été identifiés, l'appliquer sur la totalité de l'application serait trop long. Pour être appliqué à un kernel donné, il faut tout d'abord calculer la taille du jeu de données utilisé pour l'exécution, notée $\texttt{DATA}_{\texttt{size}}$. En lisant le code, il est alors possible de calculer la quantité de données minimale qui doit être transférée sur le bus mémoire. Grâce à l'étape 2, nous connaissons les performances maximales théoriques $\texttt{MEMORY}_{\texttt{peak}}$ et réelle $\texttt{MEMORY}_{\texttt{max}}$ de la microarchitecture. La durée optimale pour exécuter le kernel étudié peut alors être calculée (équation 4.4). Cette durée mesurée en seconde est notée $\texttt{TEMPS}_{\texttt{optimal}}$.

$$\texttt{TEMPS}_{\texttt{optimal}} = \frac{\texttt{DATA}_{\texttt{size}}}{\texttt{MEMORY}_{\texttt{max}}} \tag{4.4}$$

Le modèle suppose que le code utilise un algorithme parfait (utilisation de la localité des données), que la compilation du code a été réalisée avec un compilateur parfait et qu'il est exécuté sur une plateforme parfaite. L'objectif n'est pas d'atteindre exactement cette performance, mais de s'en approcher le plus possible. Généralement, lorsqu'un défaut apparaît à un des niveaux énumérés précédemment, la performance s'éloigne radicalement de la performance optimale.

4.4.5 Application des modèles Roofline et SMM

Dans l'extrait 4.4, le profil de l'exécution du benchmark `STREAM` comporte quatre fonctions utilisées pour stresser la mémoire par différents types d'accès. Nous choisissons arbitrairement de consacrer notre analyse sur un des quatre kernels : la fonction *triad* dont le code peut être vu dans l'extrait 4.5. Cette fonction est intéressante, car ce motif d'accès est très courant dans les applications HPC.

```
1 for (j=0; j < STREAM_ARRAY_SIZE; j++)
2     A[j] = B[j] + scalar * C[j];
```

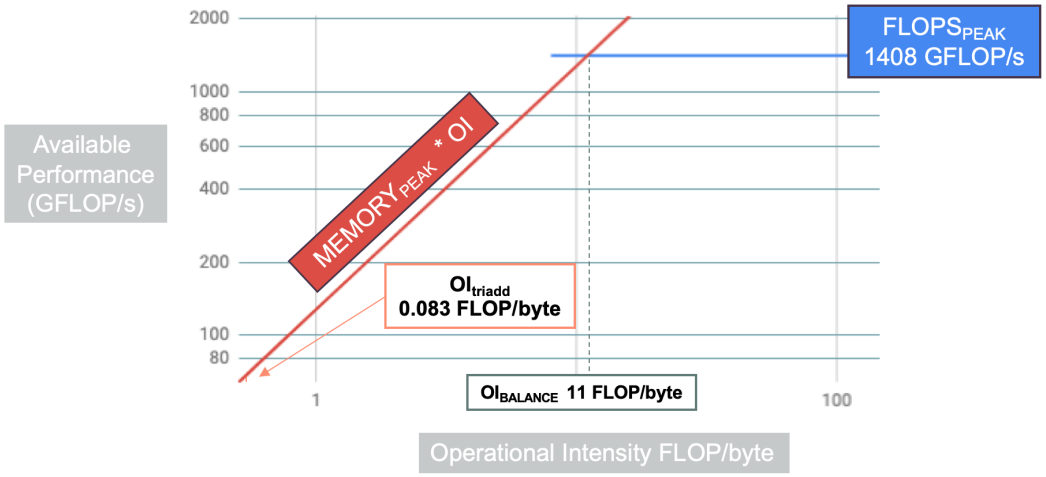


FIGURE 4.5 – Modèle du *Roofline* appliqué à la fonction *triad* du benchmark *Stream* et un processeur Xeon Skylake 6148.

Extrait 4.5 – La fonction *triad* du benchmark Stream utilise trois tableaux : deux accédés en lecture et un en écriture

4.4.5.1 Modèle du Roofline

Lors de l'étape 1, nous avons mesuré la performance mémoire maximale atteignable $\texttt{MEMORY}_{\texttt{max}}$ de 105 GB/s. L'utilisation du `Kernel Generator` nous avait permis de mesurer une performance $\texttt{FLOPS}_{\texttt{max}}$ de 1372.78 GFlop/s, proche de la performance théorique de l'architecture. Ces deux mesures permettent de construire le *toit* du modèle du Roofline présenté sur la figure 4.5.

Pour évaluer les limitations du kernel étudié, il est ensuite nécessaire de calculer son intensité opérationnelle notée $\texttt{OI}_{\texttt{kernel}}$. Lors de chaque itération de boucle, le processeur doit charger 3 éléments en double précision, soit 24 bytes, correspondant aux tableaux A, B et C de l'extrait 4.5. En effet, hors optimisation, une ligne de cache doit être chargée avant d'être écrite, même si aucune des données n'est utilisée en lecture par le processeur. À chaque itération de boucle, deux opérations doivent être réalisées, une addition et une multiplication. Cette fonction a donc une intensité arithmétique $\texttt{OI}_{\texttt{kernel}} = \frac{2}{24} = 0.083$ flop/byte. Pour comparaison, les processeurs récents ont un ratio proche de 10 $flop/byte$. Cette simple modélisation montre le déséquilibre qu'il y a entre la performance de la mémoire et celle des processeurs. Elle permet de guider le choix de la plateforme sur laquelle cette fonction devra être portée. La figure 4.5 montre l'application du Roofline à l'étude de la fonction *triad* et au processeur étudié. Cette fonction ayant une faible intensité opérationnelle ($\texttt{OI}_{\texttt{kernel}}$), ses performances théoriques mesurées en flop sont elles aussi très faibles.

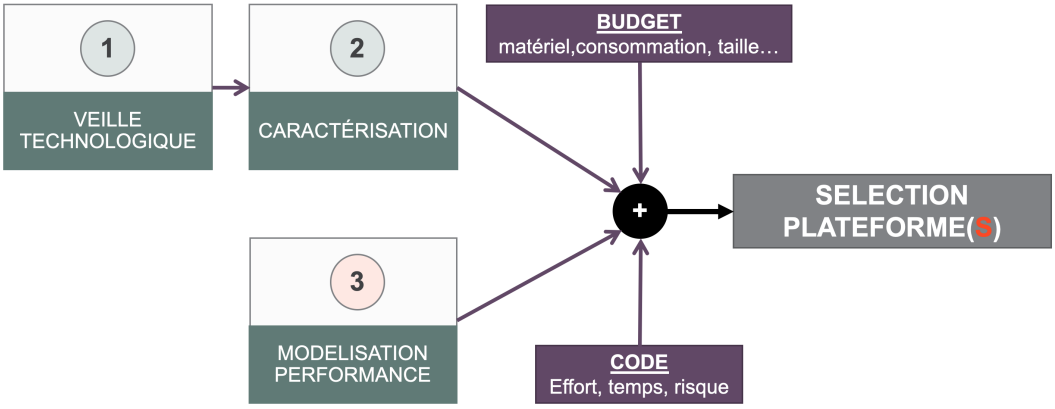


FIGURE 4.6 – L'étape 4 consiste à choisir les plateformes adaptées aux différents noyaux en fonction de plusieurs critères.

4.4.5.2 Application du Simple Memory Model

L'analyse du noyau avec le modèle du Roofline indique que sur l'architecture ciblée, la performance du code sera limitée par la performance de la bande passante. Une fois assuré que les performances de l'application sont limitées par le système mémoire, le Simple Memory Model peut être appliqué.

Dans notre expérimentation, nous utilisons trois tableaux de 19.6 GB ($3 \times 10^9 \times sizeof(double)$). Pour une exécution optimale, le bus mémoire devrait être utilisé :

— pour la lecture des tableaux B et C ;
— pour l'écriture du tableau A.

Le trafic mémoire total serait alors de 58.8 GB. En utilisant les résultats mesurés lors de l'étape 2, on peut estimer le temps optimal pour l'exécution de cette fonction : $\texttt{TEMPS}_{\texttt{optimal}} = \frac{58.8}{128} = 0.56$ seconde. Cette modélisation nous permet de projeter la performance optimale sur une architecture sans avoir à exécuter le code sur celle-ci. La valeur $\texttt{TEMPS}_{\texttt{optimal}}$ permet ensuite de vérifier que les performances maximales de l'architecture sont atteintes. Dans le cas contraire, cette valeur peut être utilisée pour quantifier le gain de performance pouvant être obtenu grâce à l'optimisation du code.

4.5 Étape 4 : Sélectionner la plateforme adaptée

Les deux premières étapes ont permis de trouver et de caractériser de potentielles architectures. À l'aide de plusieurs benchmarks, certaines caractéristiques clés des architectures ont pu être obtenues. Lors de l'étape 3, l'extraction et la modélisation des hot spots ont été faites. Cette quatrième étape consiste à réaliser le choix de l'architecture la plus adaptée pour l'exécution de chaque hot spot. Ce choix est réalisé en prenant en considération la performance des architectures, les besoins de l'application étudiée ainsi que d'autres critères (voir figure 4.6).

4.5.1 Le coût

Le prix des architectures est un élément important pour choisir la ou les architectures utilisées pour la construction de la plateforme. Cependant, utiliser le prix unitaire des accélérateurs choisis n'est pas suffisant et il est nécessaire de prendre en compte d'autres paramètres dans le calcul du prix appelé coût total de possession (TCO). Il prend en compte tous les coûts engendrés par le centre de données durant son cycle de vie : construction des bâtiments, consommation électrique, etc.

Le prix du matériel Le prix des accélérateurs et du matériel nécessaires à la construction du supercalculateur constituent une part significative dans le calcul du coût. En fonction du prix de l'accélérateur, une solution moins performante pourra lui être préférée. Les cartes FPGA sont un bon exemple d'architectures très performantes, mais peu utilisées dans les supercalculateurs. Bien que la complexité de programmation y participe, le prix des cartes FPGA est une raison majeure de leur faible utilisation dans les plateformes modernes.

La consommation électrique La consommation électrique de la plateforme finale est devenue un critère très important dans le choix du matériel. La consommation des centres de données est un réel investissement qui doit être mesuré et anticipé lors de l'achat du matériel. Un supercalculateur consommant 10 MW engendrera une facture de plusieurs millions d'euros chaque année. Le prix de l'électricité et l'enveloppe énergétique disponible pour le calcul varient avec l'emplacement choisi pour installer le centre de calcul. La majorité des centres ayant des lignes électriques déjà construites ne peuvent acheminer qu'une quantité limitée de courant. L'enveloppe énergétique disponible est alors une contrainte forte pouvant favoriser l'utilisation d'une architecture plus efficace énergétiquement.

En comparant l'intensité opérationnelle d'un kernel ($\texttt{OI}_{\texttt{kernel}}$) et l'équilibre arithmétique de l'architecture ($\texttt{EQUILIBRE}_{\texttt{architecture}}$) il est possible d'estimer la pertinence d'un accélérateur pour un noyau. Des valeurs proches indiquent que l'architecture ciblée est adaptée au code étudié et que l'accélérateur choisi aura un meilleur rendement énergétique. Un code faisant peu de calculs flottants ne nécessitera pas l'utilisation de coeurs complexes réalisant plusieurs dizaines de FLOP par cycle. Bien que non utilisés, ces composants impactent le prix de la solution, mais aussi sa consommation électrique.

La taille du centre de données Souvent les centres de données sont déjà existants et la taille disponible pour la création d'un supercalculateur ou l'ajout de nouveaux serveurs est une contrainte forte. Certaines solutions prenant plus ou moins de place pour être installées, le ratio $\frac{flop}{m^2}$ peut alors être calculé pour évaluer la densité des serveurs pour s'adapter aux contraintes du lieu. Si le bâtiment doit être construit pour accueillir le supercalculateur, son coût doit entrer dans le calcul de la solution finale.

4.5.2 La performance

D'autres critères concernant la performance et l'optimisation du code doivent être pris en compte pour choisir la ou les architectures utilisées pour accélérer l'exécution de l'application.

Performance du noyau Bien sûr, le gain de performance obtenu après le portage d'un kernel sur un accélérateur est un critère important. La performance du noyau a un impact financier, car s'il est exécuté plus rapidement, d'autres applications pourront alors accéder à la plateforme. Les clients de supercalculateurs ont généralement un budget fixe et de multiples applications à exécuter. Ils cherchent alors à optimiser l'utilisation des ressources informatiques par les différents codes. Bien que l'on considère qu'il est nécessaire de porter individuellement chaque noyau sur l'accélérateur le plus adapté, il faut aussi prendre en considération le reste de l'application, mais aussi les applications des autres utilisateurs. Si une seule des applications ne bénéficie pas des performances d'un accélérateur, il serait plus pertinent d'opter pour un accélérateur adapté à plusieurs applications exécutées sur la plateforme.

Difficulté du portage Les transformations de codes nécessaires pour porter le code d'un kernel sur une architecture doivent être évaluées. Celles-ci peuvent avoir un impact sur les performances finales (incapacité à réaliser les optimisations nécessaires) ou sur le coût (recours à des programmeurs expérimentés). Suivant l'architecture choisie, il faudra peut-être coder l'application avec un nouveau langage, utiliser de nouvelles librairies ou de nouveaux modèles de programmation. Lorsque l'implémentation d'une optimisation est décidée, le risque de ne pas parvenir à obtenir les performances espérées doit lui aussi être mesuré. Le temps optimal, noté $\texttt{TEMPS}_{\texttt{optimal}}$, pour exécuter l'application considère que l'application utilise de façon optimale l'architecture. Cependant pour atteindre ces performances, le noyau peut nécessiter l'utilisation d'optimisations complexes, pouvant être difficile à implémenter sur des applications industrielles. Il peut arriver qu'une optimisation moins performante soit préférée, car la transformation du code est plus facile. Le temps et le nombre de programmeurs nécessaires à son implémentation entrent alors aussi en considération dans le prix de la solution.

4.6 Étape 5 : Porter et optimiser le code

L'étape 4 a permis de choisir une ou plusieurs architectures pour porter les différents kernels d'une application. L'étape 5 consiste à apporter les transformations au code pour porter chaque kernel sur une architecture et à valider les performances atteintes.

4.6.1 Motivations

Suite au choix de l'architecture lors de l'étape 4, le travail de portage doit être réalisé. L'effort nécessaire pour adapter le code à cette nouvelle architecture a dû être évalué lors de l'étape précédente et peut impliquer certaines difficultés :

— Utiliser un nouveau langage de programmation ou un nouveau paradigme de programmation.
— Vérifier que les performances obtenues après le portage sont celles attendues par les prédictions.
— Avoir à sa disposition des outils compatibles avec l'architecture pour réaliser l'analyse et la validation des performances.

Dans la suite de cette section, nous montrons comment les outils développés durant ces travaux de thèse sont utilisés pour valider la performance du kernel établie lors de l'étape 3.

Développement d'une première version. Afin de s'assurer que les premières versions du code atteignent des performances acceptables, le programmeur doit rechercher les bibliothèques existantes pouvant être utilisées. En effet, les constructeurs de l'accélérateur peuvent avoir développé des bibliothèques pouvant être quasi-optimales et qui nécessiteraient une grande expertise pour être développées. Pour atteindre le maximum de performance, le code doit profiter au maximum de la parallélisation. Pour cela, le maximum de coeurs doit être utilisé, grâce aux paradigmes de programmation à mémoire partagée et mémoire distribuée. Les processeurs étant généralement superscalaires, voir section A.2.4.2, il faut essayer d'exécuter le maximum d'instructions à chaque cycle. Le dernier niveau de parallélisme est apporté par l'utilisation d'instructions vectorielles. Lorsque c'est possible, des instructions telles que les FMA doivent être utilisées. Les différentes pistes listées peuvent nécessiter l'utilisation d'un simple drapeau lors de la compilation ou au contraire nécessiter de lourdes modifications du code et des jeux de données.

4.6.2 Analyse de performance

Lors de l'étape 3, un modèle de performance du noyau a été établi (voir section 4.4.4). Une fois celui-ci porté sur la nouvelle architecture, il convient de vérifier que les performances obtenues sont proches de celles attendues par le modèle développé. La figure 4.7 propose un cheminement à suivre pour la validation et l'optimisation des performances.

4.6.2.1 Vérification du modèle de performance

La première étape consiste à vérifier que la performance atteinte par le code est proche de celle calculée par notre modèle SMM. Dans le cas contraire, cela permet de quantifier l'écart par rapport à l'optimum théorique ($\texttt{TEMPS}_{\texttt{optimal}}$).

Pour mesurer le temps d'exécution du noyau, noté $\texttt{TEMPS}_{\texttt{mesure}}$, nous proposons d'utiliser la fonction `gettimeofday ()` [Lin]. Cette fonction est disponible sur la totalité des systèmes Linux et permet de récupérer l'heure actuelle avec une précision allant jusqu'à la microseconde. Le but de la méthodologie présentée est de porter les codes sur de nouvelles architectures. Baser l'analyse de performance sur des compteurs matériels trop complexes aurait réduit la portabilité de notre démarche. Il est alors possible de mesurer $\texttt{TEMPS}_{\texttt{mesure}}$ en plaçant deux appels à la fonction `gettime()` (avant et après le noyau) présentée dans l'extrait 4.6.

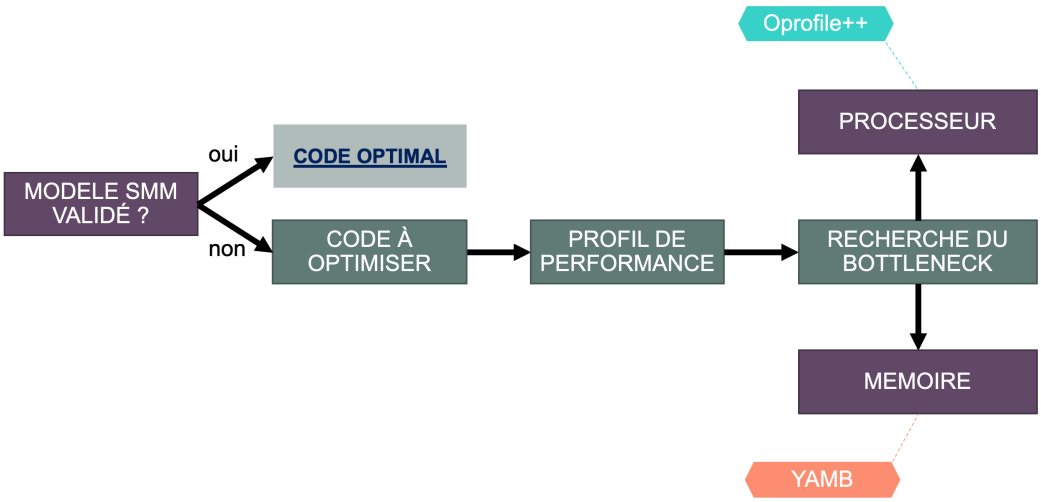


FIGURE 4.7 – Les différentes étapes à suivre pour valider la performance du noyau. Si le modèle SMM est validé, le code est optimal. Dans le cas contraire, il faut utiliser les outils adéquats pour trouver d'où vient la mauvaise performance du code (des unités de calculs ou du système mémoire

).

```
1        double gettime()
2        {
3          struct timeval tp;
4          struct timezone tzp;
5          int i;
6          i = gettimeofday(&tp,&tzp);
7          return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
8        }
9
```

Extrait 4.6 – Fonction utilisée pour obtenir l'heure actuelle avec une précision allant jusqu'à la microseconde

Si $\texttt{TEMPS}_{\texttt{mesure}}$ est proche de $\texttt{TEMPS}_{\texttt{optimal}}$, alors le noyau est proche d'avoir des performances optimales. Dans le cas contraire, l'analyse de performance doit se poursuivre pour définir la cause de cette mauvaise performance. La mesure $\texttt{TEMPS}_{\texttt{optimal}}$ permet alors d'avoir un objectif de performances à atteindre et de savoir quand le travail d'optimisation est terminé.

4.6.2.2 Profilage des performances

Lorsque la mesure de $\texttt{TEMPS}_{\texttt{mesure}}$ est inférieure à l'optimal $\texttt{TEMPS}_{\texttt{optimal}}$, le programmeur doit réaliser les modifications appropriées pour améliorer la performance du kernel étudié. Il doit pour cela avoir à sa disposition des outils lui permettant de mener à bien cette l'analyse. Utilisés de façon méthodique, les deux outils présentés dans cette thèse, `OProfile++` et `YAMB`, permettent de répondre à beaucoup de questions et mener à bien le travail d'optimisation du code. Même

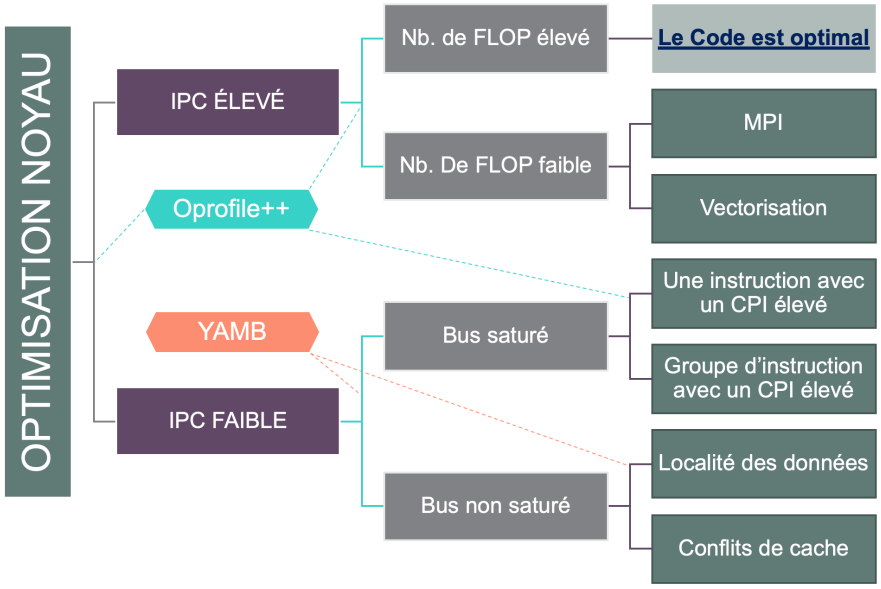


FIGURE 4.8 – Flux de travail pour l'utilisation des outils `OProfile++` et `YAMB`. En fonction des observations amenées par les outils, des réponses différentes sont proposées au programmeur pour optimiser son code.

si l'analyse par le modèle du *Roofline* a montré que la performance maximale atteignable était limitée par le débit mémoire, d'autres facteurs peuvent affecter les performances (mauvaise compilation, dépendances de données, mauvaise utilisation de la localité). Le programmeur doit alors suivre une démarche logique lui permettant d'identifier la raison de cette mauvaise performance (voir figure 4.8).

La première mesure à réaliser est celle du nombre d'instructions réalisées par cycle d'horloge (IPC) des boucles critiques du kernel. Pour cela, nous avons présenté l'outil `Oprofile++` dans la section 3.5 (voir extrait 4.7). Si l'outil indique un IPC faible, il y a de fortes chances que le système mémoire soit responsable de la mauvaise performance. Un IPC élevé indique quant à lui que le problème vient généralement du processeur.

```
1  ==================================================================
2  Analysis from the app name (horner1_long) hot spot from the symbole name (f1(
       double)) which takes 50.1878%
3  ==================================================================
4  CYCLES        INSTS       ADDRESS     disassembly
5  ------------------------------------------------------------------
6        5            11       401be0     vmovupd 0xd1b8(%rip),%ymm1
7       39            79       401bf0     vfnmadd231pd 0xd1c7(%rip),%ymm11,%ymm1
8        3            14       401bf9     vfmadd213pd %ymm2,%ymm11,%ymm1
9  ...
10     760          1652       401c3e     cmp     %eax,%edx
11      48            99       401c40     jb      401be0 <_Z8range_f1ddi+0xa0>
12 ------------------------------------------------------------------
13 LOOP from 401c40 to 401be0  size= 96 sum(cycles)= 2287
14       IPC= 2.12462 cycles/LOOP= 10.3548 flop/cycle = 1.42
15 ------------------------------------------------------------------
```

Extrait 4.7 – L'outil `OProfile++` permet d'extraire le code assembleur et d'y associer le compteur de cycles

4.6.2.3 Profilage de la mémoire

Lorsque la première analyse indique que la mauvaise performance du noyau est due au système mémoire, il est nécessaire de réaliser l'analyse de son activité. La première vérification à faire est de s'assurer que le bus mémoire est saturé. Pour y répondre, nous proposons d'utiliser l'outil `YAMB` (voir section 3.4) qui affiche l'activité du bus mémoire (lecture, écriture et utilisation totale).

Pour confirmer sa saturation, il est nécessaire d'avoir caractérisé la performance du bus mémoire lors de la première étape et d'en connaître la performance maximale. De plus, il est important de posséder une fréquence d'échantillonnage suffisamment élevée. En effet, avec une fréquence trop faible, il peut arriver que le graphique montre une saturation avec une ligne droite atteignant la performance maximale. En augmentant la fréquence et en grossissant certaines parties du graphique, il est possible de voir que le bus n'est pas totalement utilisé pendant des périodes très courtes, et qu'il est saturé quelques cycles après. Ce phénomène peut être dû à des dépendances entre plusieurs instructions. Des techniques de déroulement de boucles sont alors très efficaces si la nature de l'application le permet (voir section 3.2.3.7). Un autre facteur peut venir de la mauvaise gestion du préchargement des données. Il peut alors être intéressant de réaliser le préchargement manuellement, en démarrant les chargements de données plusieurs instructions avant qu'elles soient utilisées. D'autres techniques, telles que la fusion de différentes boucles (ou noyau) nous ont permis d'améliorer la performance de plusieurs applications.

Si le bus est bien saturé, nous utilisons le modèle SMM pour comparer les ratios de lecture/écriture avec celui calculé dans l'étape 1 (voir section 4.4.4). Un mauvais ratio peut provenir d'un nombre de lectures plus élevé que prévu. En effet, il est très rare de réaliser plus d'écriture que nécessaire. Ceci est causé par un large éventail de problèmes potentiels, par exemple, des

conflits dans le cache, la lecture de données inutiles, une décomposition incorrecte des données ou des structures de mémoire inappropriées. Pour y remédier, des techniques de *blocking* peuvent être utilisées pour améliorer l'utilisation de la localité des données. L'outil `Oprofile++` peut alors aider à trouver si des groupes d'instructions sont plus longs à exécuter que d'autres, pouvant être expliqués par une dépendance entre les données.

4.6.2.4 Profilage du processeur

Si la première analyse montrait que la mauvaise performance n'était pas due au système mémoire, il faut alors affiner l'analyse de l'exécution du code donnée par `Oprofile++` (voir extrait 4.7). La première vérification à faire est de compter le nombre d'opérations flottantes réalisées par la boucle. Pour cela, nous comptons manuellement le nombre d'opérations flottantes à partir des instructions assembleurs utilisées. Si le nombre d'opérations flottantes et l'IPC sont élevés, cela indique que le code est optimal.

Un nombre d'opérations flottantes faible associé à un IPC élevé indique que la parallélisation n'est pas correctement utilisée. Il est alors conseillé de regarder le type d'instructions exécutées. Un compilateur de mauvaise qualité aura tendance à générer de nombreuses instructions supplémentaires simples à exécuter. Cela fait augmenter l'IPC de la boucle sans en améliorer la performance. Un mauvais compilateur aura tendance à générer plus d'instructions qu'un bon compilateur pour le calcul d'adressage par exemple. La performance du code peut alors être limitée par le matériel responsable du calcul d'adresses et non par la FPU. Un autre exemple couramment rencontré est l'exécution d'instructions de la bibliothèque interface de passage de messages (MPI) utilisée pour synchroniser les processus. Celles-ci sont exécutées à chaque cycle pour vérifier l'état des autres processus, faisant augmenter l'IPC de la boucle. Il faut alors utiliser des outils tels que *Extrae* [Rod], *Paraver* ou Vampire [Nag+96 ; BW13] pour vérifier que la répartition du travail est bien réalisée. Un noeud de calcul ayant un matériel défectueux affecte alors les autres serveurs, et de nombreux cycles sont perdus dans ces instructions de synchronisation. Les pannes sont très difficiles à identifier sans les outils adaptés, car ils peuvent venir d'une multitude d'endroits : barrette mémoire défectueuse, surchauffe d'un processeur ou d'un disque cassé. Si la boucle contient beaucoup d'instructions de calcul à virgule flottante, il faut vérifier qu'il s'agit d'instructions vectorielles les plus larges possible.

Il est donc important de ne pas baser son analyse seulement sur la lecture de l'IPC, mais de vérifier que les instructions exécutées sont des instructions de calculs flottants. Les transformations du code nécessaires pour y parvenir peuvent être difficiles à réaliser et demander l'utilisation d'un autre algorithme ou de restructurer le jeu de données.

4.6.3 Application au benchmark STREAM

Nous appliquons notre méthodologie sur le benchmark `STREAM` exécuté sur un processeur Intel Skylake. Cet exercice nous permet de montrer que même sur un code aussi simple, apparemment optimal, notre analyse nous permet de comprendre sa performance et de l'optimiser.

4.6.3.1 Vérification du modèle SMM

Nous avons montré précédemment que la performance du kernel était limitée par la bande passante mémoire. Nous avons appliqué notre modèle de performance SMM et calculé $\texttt{TEMPS}_{\texttt{optimal}}$ égale à $\frac{58.8}{128} = 0.56$ seconde. Suite à l'exécution du noyau étudié (kernel *triad*), et à l'utilisation de la fonction présentée dans l'extrait 4.6, nous mesurons $\text{TEMPS}_{mesure} = 0.79$ seconde. L'application n'est donc pas optimale. Ceci est également validé par le résultat du benchmark `STREAM` qui annonce une bande passante mémoire de 80,13 GB/sec, inférieure à la performance du bus mémoire mesurée à l'aide de `DML_MEM`. Dans l'étape 2, nous avions mesuré un débit maximal $\texttt{MEMORY}_{\texttt{max}}$ de 105 GB/s.

Pour comprendre ce résultat, nous avons utilisé `YAMB` pour vérifier que le bus mémoire était bien saturé. Le graphique de la figure 4.9 nous montre que le bus mémoire est utilisé au maximum de son potentiel de 104 GB/s. La saturation du bus n'est donc pas la cause de la mauvaise performance du noyau. Il est alors nécessaire de regarder les rapports lecture/écriture. Lors de l'étape 3, nous avions calculé un ratio optimal pour ce noyau d'une écriture pour deux lectures. La figure 4.9 montre la répartition mesurée lors de l'exécution : 26 GB/s en écriture pour 78 GB/s en lecture, pour un total de 104 GB/s, ce qui correspond à un ratio de 1 écriture pour 3 lectures. Ce ratio exact d'une écriture pour trois lectures n'est pas une coïncidence. Un tableau complet est lu alors qu'il ne devrait pas l'être.

4.6.3.2 Optimisation du noyau *triadd*

Les deux tableaux B et C doivent nécessairement être lus une fois. La lecture supplémentaire provient du tableau en écriture. Ce comportement est dû au processeur qui, avant d'écrire une donnée, charge la ligne de cache correspondante pour la mettre à jour. Cependant, le noyau étudié a la particularité d'écrire toute la ligne de cache. Il n'est donc pas nécessaire de charger les données initialement présentes. Une option peut être utilisée avec le compilateur Intel (ICC) pour permettre au compilateur d'éviter ce chargement inutile : `-qopt- streaming-stores=always`. Cette option permet au CPU d'écrire toute la ligne de cache sans avoir à la charger.

4.6.3.3 Performance du noyau optimisé

Nous avons compilé `STREAM` avec l'option `-qopt- streaming-stores=always` et mesuré le temps et la bande passante de la même manière que pour la première exécution. Les résultats sont résumés dans le tableau 4.1. Le temps passé dans le noyau est de 0,59 seconde, beaucoup plus proche du maximum théorique de 0,56 seconde. En analysant la bande passante mémoire, on constate que le rapport lecture/écriture est maintenant de 2 pour 1, les données échangées étant de 34 GB/s en mode écriture pour 68 GB/s en mode lecture. La bande passante totale atteint 104 GB/s proche de $\texttt{MEMORY}_{\texttt{max}}$.

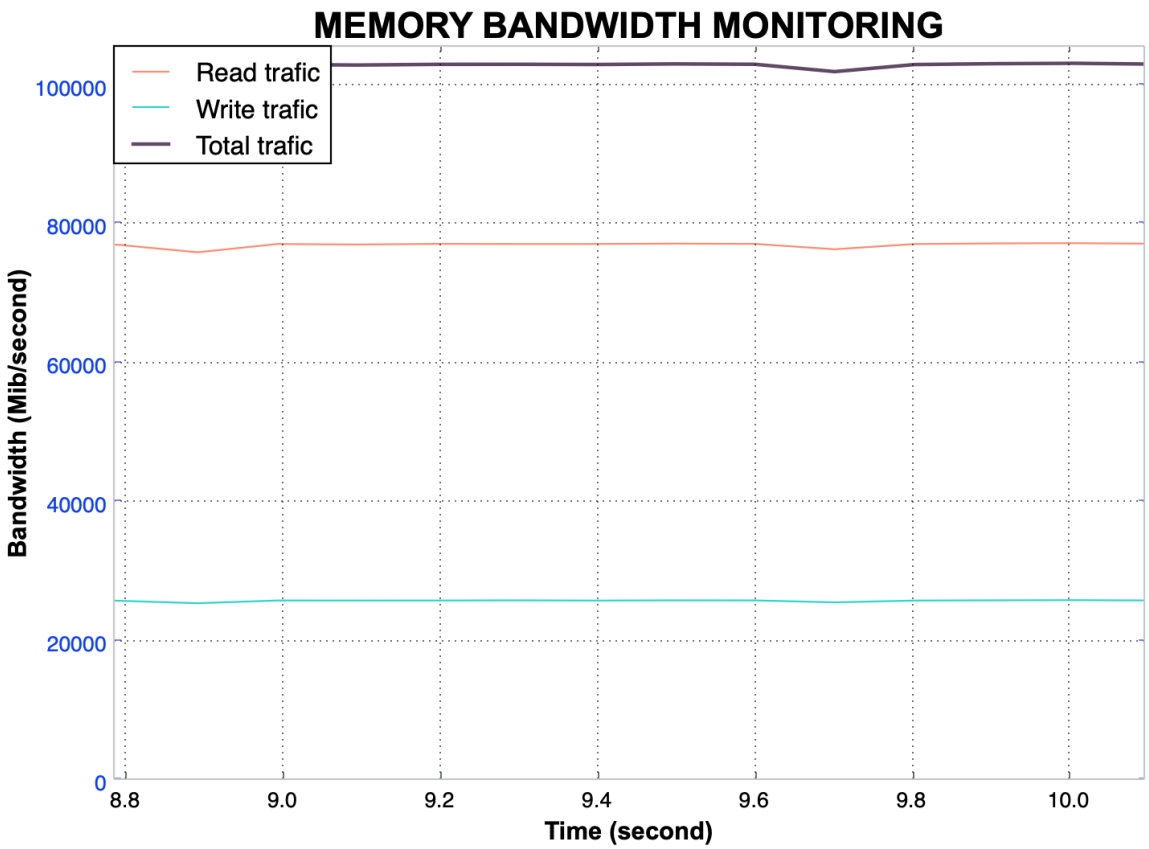


FIGURE 4.9 – Profil mémoire donné par l'outil `YAMB` pour plusieurs exécutions de noyau *triadd* du benchmark `STREAM`.

4.7 Conclusion et perspectives

Dans ce chapitre, nous avons présenté une méthodologie permettant de faciliter l'utilisation d'architectures hétérogènes dans les plateformes de calcul haute performance. Afin de motiver la réalisation de ce travail, nous avons commencé par rappeler l'importance de l'utilisation de matériels hétérogènes dans les supercalculateurs. Nous avons aussi relevé que cette hétérogénéité n'en était qu'à ses débuts avec l'utilisation des GPU. Pour répondre à ce besoin, nous avons détaillé les cinq étapes permettant d'utiliser les outils présentés dans le chapitre précédent.

Dans un premier temps, nous avons discuté de la recherche et de la caractérisation de ces nouvelles plateformes. Pour cela, nous avons résumé les principales caractéristiques qu'il est nécessaire de posséder pour réaliser un premier tri. Nous avons ensuite présenté comment les outils de benchmark pouvaient être utilisés pour caractériser la performance et le comportement des microarchitectures.

Dans un second temps, nous nous sommes intéressés à l'application et à la nécessité de posséder un outil permettant d'identifier ses noyaux de calculs et de modéliser leur performance.

	$\text{TEMPS}_{mesure}(s)$	Stream	YAMB_{read}	YAMB_{write}	YAMB_{total}
Version originale	0.79	80.13	26	78	104
Version optimisée	0.59	101.84	35	69	104

Tableau 4.1 – Performance du noyau *triadd* du benchmark `STREAM` avant et après optimisation. Le temps, mesuré en secondes, après optimisation est proche de l'optimum théorique $\texttt{TEMPS}_{\texttt{optimal}}$ calculé à 0,56 sec. L'utilisation du bus mémoire est la même entre les deux versions du code. L'option de compilation optimise le rapport lecture/écriture (GB/s) et améliore les performances du code de 25%.

La performance de la majorité des applications de calcul haute performance étant limitée par celle du bus mémoire, nous avons présenté un modèle de performance simple, basé sur la le ratio de lecture/écriture et la taille des jeux de données.

Après avoir caractérisé les architectures et modélisé les noyaux des applications, nous avons discuté dans une quatrième étape des principaux facteurs à étudier, lors du choix d'une nouvelle architecture. La performance n'étant qu'un critère parmi d'autres tel que le coût TCO de la solution, la consommation électrique ou la difficulté des transformations nécessaires.

Enfin, nous avons présenté une dernière étape permettant de valider le modèle de performance précédemment réalisé. Lorsque le code n'atteint pas les performances espérées, nous proposons un cheminement pour trouver d'où vient le problème et comment atteindre les performances souhaitées.

Pour illustrer la méthodologie, nous avons présenté pour chaque étape son application sur l'application Stream et sur un processeur Intel Skylake. Cet exercice nous permet de montrer que même pour un code aussi simple et en apparence optimisée, l'approche et les outils utilisés permettent de comprendre et d'optimiser ses performances.

Perspectives

Lors de l'étape cinq, nous avons proposé un cheminement à suivre pour comprendre la mauvaise performance d'un code et des pistes à suivre pour l'améliorer. Cette étape demande une très bonne connaissance du matériel (microarchitectures, stockage, réseaux), mais aussi du logiciel (système d'exploitation, programmation à mémoire partagée et distribuée). L'expérience acquise durant ces travaux de thèse nous a montré que chaque cas était différent et qu'il fallait beaucoup d'expérience pour identifier les goulots d'étranglement et comprendre les mauvaises performances. Un travail préliminaire a été réalisé, permettant de guider l'utilisateur dans son analyse en expliquant quand utiliser chaque outil et comment transformer le code à partir des informations extraites.

Chapitre 5

Conclusion générale

Le domaine du Calcul Haute Performance (HPC) est en pleine croissance et les utilisateurs de supercalculateurs ont un besoin insatiable de puissance de calcul (conduite autonome, recherche sur le climat, villes intelligentes). Des plateformes plus puissantes doivent être développées pour être capables d'analyser le tsunami de données générées par les objets connectés. Ces analyses nécessitent l'utilisation d'algorithmes complexes ne pouvant être exécutés en un temps raisonnable que par des supercalculateurs 10 fois plus puissants que ceux actuellement utilisés.

Contexte de la thèse

Depuis plus de 50 ans, les systèmes d'information participent activement à l'amélioration des connaissances scientifiques et à l'évolution de nos sociétés. Ils sont également des vecteurs indispensables pour augmenter la compétitivité et l'innovation des entreprises grâce à l'utilisation de plateformes de calculs puissantes nommées supercalculateurs. Ceux-ci sont utilisés pour exécuter des applications complexes provenant de différents domaines. Que ce soit dans le domaine de la physique, de la santé ou pour développer de nouvelles intelligences artificielles, les supercalculateurs jouent un rôle décisif dans l'évolution de notre société.

La performance des supercalculateurs a pu évoluer constamment pendant plusieurs dizaines d'années grâce aux améliorations technologiques : augmentation du nombre de transistors (loi de Moore) et de la fréquence des processeurs, complexification de la microarchitecture). Cependant, ces leviers ne sont plus disponibles pour permettre d'élaborer des plateformes plus puissantes. La principale contrainte étant celle de l'énergie, qui, pour des raisons de coût, de faisabilité et de respect de l'environnement, ne permet plus de poursuivre la stratégie consistant à augmenter le nombre de serveurs pour augmenter linéairement la performance des supercalculateurs.

Avec la fin de la validité de la loi de Moore, nous devons trouver d'autres moyens d'améliorer la performance de ces plateformes. En effet, l'incapacité des technologies actuelles à réaliser efficacement des calculs et l'explosion du nombre de données à traiter nous obligent à repenser les technologies utilisées et l'architecture des systèmes informatiques. Une solution consiste à utiliser des architectures spécialisées pour le traitement optimisé de certains algorithmes. Ces architectures vont pouvoir être développées grâce à l'utilisation de nouvelles technologies (*storage class memory* (SCM), mémoire 3D, interconnexion photonique). Ces ruptures technologiques sont telles qu'elles permettront d'améliorer la performance des architectures de plusieurs facteurs. Les gains de performances ne viendront pas seulement de l'utilisation d'accélérateurs puissants, mais de leur diversité et de la capacité des programmeurs de bien les

utiliser. Pour une même application, plusieurs accélérateurs spécialisés seront souvent nécessaires (ASIC, FGPA, DSP, GPU). Grâce au développement d'un protocole de communication universel nommé Gen-Z, ces différentes architectures pourront être interconnectées dans un même système pour travailler ensemble à l'exécution d'une application de façon efficace.

La capacité à produire plus de résultats avec des ressources limitées devient une nécessité absolue dans un monde hautement compétitif. Cependant, pour réussir, ces accélérateurs, basés sur une technologie de rupture, devront être entièrement caractérisés pour prédire de manière fiable le gain de performance de l'application.

Afin d'extraire une large part des performances disponibles de ces architectures (très différentes de celles utilisées actuellement), il est impératif de réaliser un travail d'analyse et d'optimisation de performance. Malheureusement, l'évolution des microarchitectures tous les deux ans (modèle Tic-Toc d'Intel) ne nécessitait pas une telle approche. Par conséquent, nous constatons que les outils nécessaires à ce travail sont rares. Sans ce travail de caractérisation et d'analyse de performance, ces architectures sont potentiellement condamnées, car peu d'applications pourront y être exécutées efficacement.

5.1 Problématique

5.1.1 Problématique de la thèse

Afin de proposer les meilleures architectures (performance, efficacité énergétique, prix, etc.) pour le développement de nouveaux supercalculateurs, une entreprise comme HPE doit être capable d'identifier et de caractériser ces nouvelles plateformes. À cet effet, nous proposons dans ce travail de thèse une suite de logiciels de caractérisation et d'analyse des performances qui, avec la méthodologie appropriée, permettent de sélectionner la meilleure plateforme et d'extraire une part significative de ses performances. En rendant les sources des logiciels disponibles, nous voulons sensibiliser les utilisateurs à cette démarche, ce qui permettra de poursuivre le travail de caractérisation du nombre croissant de nouvelles plateformes.

Cependant, la caractérisation d'architectures et l'analyse de performance d'applications sont deux domaines très complexes, nécessitant de nombreuses connaissances. Pour permettre le développement des outils manquants, nous avons décrit ces deux difficultés. Afin de faciliter le travail de futurs programmeurs, nous les présentons ces deux études dans les annexes de ce document :

— L'Annexe A couvre l'origine et l'évolution de la microarchitecture des processeurs. Nous présentons les fonctionnalités clés des processeurs modernes qu'il est nécessaire de connaître pour comprendre la performance des applications : *pipelines*, instructions vectorielles, etc. Nous étudions plus précisément la hiérarchie mémoire qui est la ressource critique de nombreuses applications.

— L'Annexe B présente les compteurs matériels. Ces registres spéciaux du processeur permettent de suivre l'exécution d'une application en mesurant le nombre d'évènements logiciels et matériels. La programmation des compteurs est très difficile et nécessite le

recours à des codes bas niveaux. Plusieurs méthodes peuvent alors être employées et demandent une bonne expérience pour être mises en oeuvre.

5.2 Contributions principales de la thèse

5.2.1 État de l'art du domaine du calcul haute performance

Afin, d'introduire les contributions apportées dans ce travail de thèse, nous avons débuté dans le chapitre 2 par la réalisation d'un large état de l'art du domaine du Calcul Haute Performance. Nous avons ainsi commencé par rappeler les fondements du calcul scientifique et de la programmation parallèle. Afin de mieux estimer les performances actuelles des supercalculateurs, nous avons étudié l'évolution de celles-ci au cours des 20 dernières années. Cette étude s'appuie sur le classement du Top500 qui présente, deux fois par an, les 500 supercalculateurs les plus puissants de la planète. Ainsi, nous avons remarqué un ralentissement dans l'évolution de leurs performances depuis 2012 et nous en avons expliqué les principales raisons.

Après avoir discuté de la nécessité d'accéder à des plateformes de calcul plus puissantes, nous avons étudié les principaux défis empêchant leur développement : le coût, la consommation énergétique, la complexité des microarchitectures, l'explosion du nombre d'architectures disponibles, le besoin de prendre des décisions rapidement et le manque d'expertise. La principale contrainte, liée à l'énergie, nous a ensuite conduit à présenter les opportunités technologiques qui nous permettront de relever ces défis : la présence de nombreux investissements, les nouvelles technologies mémoire (Storage Class Memory (SCM)), les nouvelles technologies d'interconnexion (photoniques) et le protocole universel Gen-Z. L'ensemble de ces nouvelles technologies sont utilisées pour développer des architectures innovantes. Ces architectures, très différentes de celles que nous utilisons actuellement, doivent être caractérisées afin de connaître leurs forces et leurs faiblesses : capacité de calcul, débit, latence et capacité de la hiérarchie mémoire, etc. Il est ensuite nécessaire de connaître les besoins des applications afin de sélectionner les architectures les plus adaptées pour son exécution.

Les outils existants qui permettent la caractérisation d'architectures et l'analyse de performance d'applications ont été présentés à la fin du chapitre 2. Nous avons pour cela fixé certains critères permettant de les sélectionner. Leur code source doit être libre d'accès pour pouvoir être adaptées à de nouvelles architectures. Leur utilisation doit être la plus simple possible, et répondre à des questions précises. Enfin, ces outils doivent être adaptés aux environnements de calcul hautes performances : possibilité de suivre une partie de l'exécution, réduction de la nécessité de posséder des droits spéciaux (*root*) et facilité d'installation et d'utilisation. Cette étude nous a permis de mettre en évidence la nécessite de développer de nouveaux outils :

1. Un outil permettant de caractériser finement les unités arithmétiques et logiques responsables de l'exécution d'opérations à virgule flottante (FLOP) ;
2. Un outil permettant de caractériser l'ensemble des niveaux de la hiérarchie mémoire, matériel critique pour la performance des applications utilisant des motifs d'accès mémoire par strides ;

3. Un outil permettant de suivre l'activité du bus mémoire qui est la ressource critique des architectures actuelles ;
4. Un outil permettant d'extraire et de caractériser les zones de codes responsables de la majorité du temps d'exécution des applications HPC.

5.2.2 Développement de quatre logiciels et d'une méthodologie

Le point central de ce travail de thèse est le développement de quatre outils permettant la caractérisation d'architectures et l'analyse de la performance d'applications. Contrairement aux outils existants, notre approche réside dans l'utilisation d'outils simples répondant à une question précise. Pour aider à leur utilisation, une méthodologie a ensuite été présentée. Les quatre principaux logiciels développés sont présentés dans le chapitre 3 :

1. La section 3.2 présente le benchmark mémoire `DML_MEM`. Ce code permet de caractériser les différents niveaux de la hiérarchie mémoire et de mesurer sa performance lors de l'exécution d'applications utilisant des accès mémoires par sauts (stride). La performance de ces applications dépend alors essentiellement de la capacité de ces systèmes à comprendre les motifs d'accès et à anticiper les accès mémoire (prélecteur mémoire). Le benchmark permet de tester différentes tailles de saut et de jeux de données. Ainsi, nous avons pu caractériser plusieurs aspects du matériel : taille d'une ligne de cache, performance de différentes tailles de sauts, performance du préchargement mémoire, optimisation de déroulement de boucle, performance de l'utilisation de pages larges ou encore l'impact de la fréquence et du nombre de coeurs utilisés sur le débit mémoire atteignable.
2. La section 3.3 présente le générateur de benchmarks `Kernel Generator`. Ce code permet de caractériser les unités de calcul arithmétique et plus précisément la partie responsable de l'exécution des instructions de calculs sur des nombres à virgule flottante (unité de calcul à virgule flottante (FPU)). Le benchmark génère un code assembleur correspondant aux opérations voulues par l'utilisateur (addition, multiplication, FMA (fused multiply-add)). Différentes tailles d'instructions sont supportées (SSE, AVX2, AVX512) ainsi que les précisions simple et double. Ce code nous a alors permis de vérifier les performances théoriques des unités de calculs (fréquence soutenable, performance atteignable), mais aussi de caractériser le système d'exécution dans le désordre de processeurs modernes. Enfin, nous avons pu expliquer les performances atteintes par le benchmark HPL sur un modèle de processeur Intel, alors deux fois supérieures à celles attendues.
3. La section 3.4 présente l'outil de suivi d'activité mémoire `YAMB`. Cet outil établit le profil de chaque contrôleur de mémoire en mesurant séparément le nombre de transactions (lecture et écriture) et le nombre de défaut de cache (miss) dans le dernier niveau de cache (LLC). Pour corréler l'activité du bus avec les parties du code qui en sont responsables, le graphique peut être très facilement annoté, depuis le code source de l'application, avec une API C/C++/Fortran.
4. La section 3.5 présente l'outil d'analyse `Oprofile++` permettant d'extraire et de caractériser les zones de codes responsables de la majorité du temps d'exécution de l'appli-

cation. L'outil permet de désassembler le code, d'isoler les points chauds (hot spots) et de mesurer leur IPC (Instruction Par Cycle). Il est alors possible de quantifier toutes les possibilités d'amélioration significatives. Cette étape est essentielle pour les décideurs qui peuvent prévoir avec précision le bénéfice potentiel d'une architecture par rapport à l'investissement nécessaire à la transformation du code.

Enfin, nous avons présenté dans le chapitre 4 une méthodologie permettant à un développeur de trouver l'architecture la plus efficace pour son application. Cette méthodologie en 5 étapes décrit comment les outils développés dans le chapitre 3 doivent être utilisés. L'étape 1 et 2 permettent de trouver et de caractériser de nouvelles architectures à l'aide de benchmarks. L'étape 3 s'intéresse à la modélisation des performances d'une application. Grâce à ces trois étapes, il est alors possible de sélectionner une architecture candidate dans l'étape 4. Cette sélection est réalisée en prenant en compte différents facteurs : coût, énergie, performance, etc. Dans l'étape 5, nous avons expliqué le travail permettant de valider les performances obtenues, et avons discuté des démarches à suivre lorsque les performances atteintes s'éloignaient des performances théoriques.

5.2.3 Les résultats obtenus

Les outils et la méthodologie développés durant ces travaux de thèse ont permis d'obtenir plusieurs résultats.

Les outils de caractérisation sont régulièrement utilisés par l'équipe avant-vente d'HPE, responsable de réaliser les benchmarks des applications clients. Récemment, ces outils ont permis de comprendre et d'isoler un bug majeur dans une nouvelle architecture prometteuse (alors inconnue par le fabricant) ne délivrant pas la performance attendue. Cette architecture devait alors être utilisée pour répondre à un grand appel d'offres et a pu être abandonnée avant de réaliser les transformations de codes sur l'application du client.

Le benchmark `Kernel Generator` qui permet de caractériser les unités de calcul en virgule flottante (FPU) a été ajouté au logiciel `HPE Confidence`. `HPE Confidence` permet à HPE de valider le bon fonctionnement d'un supercalculateur lors de sa livraison. Grâce à cet ajout, `HPE Confidence` vérifie les bonnes performances de calcul de chaque processeur et permet d'isoler des composants défectueux.

Les outils d'analyse de performance ont été utilisés pour remporter le `Hackaton du HPC` [Pou+18], un concours d'optimisation d'applications. Ce concours, organisé par GENCI (*Grand Équipement National de Calcul Intensif*) et sponsorisé par Intel, avait pour objectif d'optimiser une application de calcul distribué MPI pour la dynamique de systèmes particulaires. En utilisant les outils d'analyse de performance et en appliquant les optimisations adaptées, une accélération d'un facteur 10 a pu être réalisée permettant d'obtenir le prix de la meilleure optimisation [1].

Grâce à la méthodologie développée durant ces travaux de thèse, plusieurs appels d'offres ont pu être remportés. L'un d'entre eux fut lors de l'appel d'offres d'un client stratégique qui

1. Résultats du Hackaton du HPC - https://hackathon-hpc.sciencesconf.org/resource/page/id/6

autorisait le choix de plateformes spécialisées et la transformation du code. Grâce à notre méthodologie, la performance maximale théorique de l'algorithme pour la plateforme sélectionnée a été atteinte, permettant ainsi d'accélérer l'application d'un facteur 12.

Durant ces travaux de thèse, nous avons réalisé plusieurs formations pour des clients afin de les sensibiliser à cette approche et à la nécessité de posséder les outils adéquats pour réaliser le travail d'optimisation ou de portage de code.

5.2.4 Difficultés

Au moment de la publication de ce manuscrit, la grande majorité des outils a permis d'obtenir des résultats sur les architectures `x86`. Nous nous sommes appuyés sur ces architectures pour développer nos outils. En effet, l'expérience et les connaissances de ces architectures nous ont permis de développer et vérifier les résultats produits. Comme expérimenté dans le chapitre 4, nous avons utilisé ces architectures comme démonstrateur afin d'illustrer l'utilisation des outils et de notre méthodologie. Concernant l'étude d'architectures innovantes, un accélérateur a pu être caractérisé. Néanmoins, les résultats correspondants ne peuvent pas être publiés pour des raisons de confidentialité. Des architectures ayant un potentiel pour l'exécution d'application sont développées et ce travail de caractérisation doit être poursuivi.

Nous avons développé ces outils en favorisant au maximum leur portabilité. Pour cela, nous avons dû réduire au minimum le nombre de compteurs matériels utilisés. En effet, la compatibilité des compteurs d'évènements est très faible entre différentes architectures. Pour assurer la plus grande portabilité des outils de suivi de performance, nous avons choisi de nous appuyer sur la couche de programmation de compteurs maintenue par le noyau Linux (`Perf Events`). Actuellement, le benchmark du `Kernel Generator` ne permet de caractériser que les architectures utilisant des jeux d'instructions x86. Cependant, son développement a été réalisé pour faciliter l'ajout de nouvelles instructions, mais aussi d'un langage assembleur différent.

5.3 Conclusion

Ce travail de thèse présente le domaine du Calcul Haute Performance et discute du besoin de posséder les outils adéquats pour permettre l'utilisation de nouvelles architectures de calculs dans les prochaines générations de supercalculateurs exascale. Ce travail est fondamental pour pouvoir sélectionner les bonnes architectures et exécuter les applications de façon la plus optimisée possible. Cependant, l'évolution permanente des performances des processeurs a placé le domaine de la caractérisation d'architectures et de l'analyse de performance a longtemps en second plan. Face à la complexité du domaine du HPC et celle du travail à réaliser, ce manuscrit s'est voulu pédagogique pour partager le maximum de connaissances acquises durant la thèse. Pour cela, nous avons commencé dans le chapitre 1 par rappeler les principes fondamentaux des supercalculateurs (simulation numérique, programmation parallèle). Nous nous sommes ensuite intéressés aux défis à relever pour développer les prochaines générations de supercalculateurs.

Afin de tirer parti des nouvelles architectures, nous avons motivé le besoin de posséder les outils permettant de les caractériser, mais aussi ceux permettant de suivre leur activité. Dans le chapitre 3 nous avons présenté nos principales contributions qui sont le développement de deux outils de caractérisation d'architectures et deux outils d'analyse de performance. Les logiciels développés sont simples et permettent de répondre à quelques questions précises. Leur efficacité réside dans la logique de leur utilisation. Afin d'utiliser au mieux ces outils, nous avons décrit une méthodologie dans le chapitre 4. Celle-ci facilite le travail de l'équipe avant-vente d'HPE pour trier et sélectionner les architectures potentielles pour ses clients. Les outils développés permettent de caractériser les architectures, aider au portage et à l'optimisation des codes.

La réalisation de cette thèse dans un milieu industriel a beaucoup apporté à ce travail, notamment par la proximité avec l'équipe en contact direct avec les clients de supercalculateurs. Cette immersion a permis de prendre en compte les contraintes réelles des acheteurs de supercalculateurs qui peuvent être très différentes de celles rencontrées dans le milieu académique. Ce travail s'est révélé être au coeur de la stratégie d'HPE, à savoir l'optimisation de l'utilisation des ressources utilisées dans les centres de calculs. En effet, cela s'est confirmé à plusieurs occasions, notamment avec l'intégration du `Kernel Generator` au logiciel `HPE Confidence`, ce dernier étant une référence interne pour la validation des performances des fermes de calcul scientifique. Aussi, nous travaux ont été sélectionnés pour être présentés aux communautés techniques avant-vente lors de diverses conférences internationales(`HPE TSS`, `HPE TES`). Ces communautés sont en interaction directe avec les clients, ce qui a confirmé une réelle demande sur le terrain pour une méthodologie et des outils d'optimisation. Nos travaux ont également été sélectionnés par un comité d'experts pour une conférence se tenant chaque année au sein d'HPE, et mettant en compétition l'ensemble des travaux techniques réalisés chez HPE au niveau mondial. Il s'agissait de l'édition 2020 du HPE Tech Con. HPE Tech Con est un évènement qui a un impact décisionnel sur les choix des innovations technologiques pour le futur de l'entreprise. Le but de cet évènement est donc d'alimenter une réflexion globale en sélectionnant les propositions les plus novatrices venant des différentes équipes internes, tous domaines confondus. Au-delà de la pertinence technique des soumissions, ce comité composé d'experts reconnus en interne et en dehors d'HPE, évalue la maturité industrielle de la solution proposée, et exige des preuves tangibles d'avancement des travaux. Un peu plus de 500 papiers sont soumis chaque année dans le cadre de cette initiative. Notre travail a fait partie des 16 propositions sélectionnées cette année, et nous avons été invités à le présenter à l'ensemble des équipes de R et D d'HPE, lors de la conférence qui s'est tenue en février 2020 à Orlando (USA).

Perspectives

Ce travail constitue le début d'un projet de plus grande envergure visant à caractériser et analyser la performance des applications sur de nombreuses architectures. La prochaine étape est de poursuivre le développement de ces outils pour supporter de nouvelles architectures, mais aussi pour ajouter certaines fonctionnalités aux outils existants :

— **DML_MEM** : ajouter une fonctionnalité permettant de détecter automatiquement un

problème de la microarchitecture en exécutant de multiples combinaisons {`taille de saut`, `taille de jeu de données`},

— **Kernel Generator** : permettre l'utilisation de nouvelles instructions vectorielles (rotation, décalage...) pouvant être de tailles différentes dans le même microbenchmark,

— **Oprofile++** : adapter une partie du code pour réutiliser certaines fonctionnalités proposées par l'outil `perf` permettant de faire la correspondance entre les instructions assembleur et leur adresse mémoire.

— **Méthodologie** : utiliser des modèles de performances plus complexes existant dans la littérature.

Notre prochain objectif est d'appliquer notre méthodologie pour la caractérisation de plateformes telles que les processeurs AMD, ARM ou les accélérateurs NEC. Ce travail sera réalisé pour répondre à un appel d'offres pour la société ARAMCO (recherche pétrolière). Pour cela, un générateur de benchmarks générant des microbenchmarks de code de stencil sera créé permettant de caractériser les architectures pour ces codes. Un second travail consiste à pouvoir exécuter les benchmarks à intervalles réguliers sur un supercalculateur permettant de générer des traces. Grâce à des méthodes statistiques et d'apprentissage machine, ces données nous aideront à découvrir et anticiper les pannes matérielles. Enfin, comme la réussite de ce projet repose sur l'aide de la communauté, nous avons prévu de poursuivre la formation et la sensibilisation à cette démarche auprès de nombreux clients industriels ainsi que des spécialistes avant-vente HPC.

Annexes

Annexe A

Architecture des processeurs

Sommaire

A.1 Le circuit logique . . . 226
- A.1.1 Les transistors . . . 226
- A.1.2 Les portes logiques . . . 226
- A.1.3 Algèbre de Boole . . . 227
- A.1.4 Circuits logiques . . . 227
- A.1.5 Les mémoires RAM . . . 229
- A.1.6 Évolutions des transistors . . . 230

A.2 L'architecture . . . 232
- A.2.1 Architecture de processeur : origine et évolutions majeures . . . 233
- A.2.2 Améliorer l'efficacité de l'exécution . . . 237
- A.2.3 Accélérer l'exécution des instructions . . . 239
- A.2.4 Exécuter les instructions en parallèles . . . 246

A.3 Hiérarchie mémoire . . . 254
- A.3.1 Mémoire principale . . . 255
- A.3.2 La hiérarchie mémoire . . . 258
- A.3.3 Caches . . . 264

A.4 Mémoire virtuelle . . . 272
- A.4.1 La pagination . . . 274
- A.4.2 Memory Management Unit (MMU) . . . 277

L'objectif de cette annexe est de regrouper et présenter les principaux concepts techniques et technologiques qu'il a été nécessaire d'assimiler pour réaliser ce travail de thèse. En regroupant ces connaissances dans ce manuscrit, nous souhaitons faciliter le travail de futurs étudiants ou de futurs programmeurs souhaitant s'intéresser de plus près à la microarchitecture des processeurs. Pour cela, le chapitre regroupe les concepts étudiés en les empilant progressivement à la manière du modèle OSI [DZ83] :

- Niveau 1 - Le circuit logique (section A.1)
- Niveau 2 - Les processeurs
  - L'architecture (section A.2)
  - La hiérarchie mémoire (section A.3)

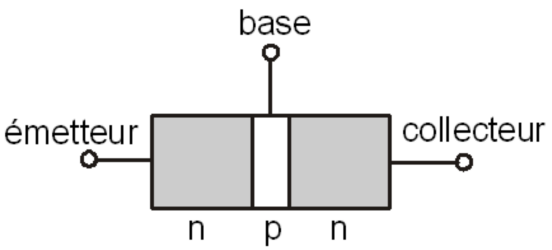
— Niveau 3 - Le système d'exploitation
  — La mémoire virtuelle (section A.4)

A.1 Le circuit logique

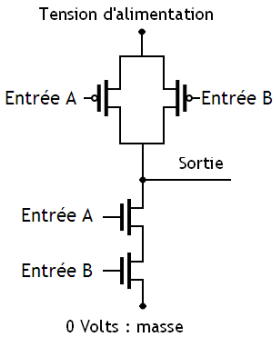
Le niveau plus bas abordé dans cette annexe est celui des transistors qui sont les composants de base de tout système électronique.

A.1.1 Les transistors

Le premier transistor a été mis au point par des chercheurs des Laboratoires Bell en 1947 [BB48], dont la découverte avait été réalisée quelques années auparavant [Edg30]. Ils apparaissent alors comme une révolution face aux tubes électroniques utilisés jusque-là, plus rapides, plus légers et plus robustes. Un transistor est un composant électronique qui utilise trois bornes : la base, l'émetteur et le collecteur (voir figure A.1a). Le collecteur est relié au fil d'où vient la tension et correspond à la sortie du transistor. L'émetteur est lui relié à la masse (tension 0 volt). La base établit la connexion entre le connecteur et l'émetteur, en fonction de la tension qui lui est appliquée. En appliquant une tension faible à la base, le courant entre le collecteur et l'émetteur est possible. Sans aucune tension appliquée à la base, le passage du courant n'est pas possible. Pour les personnes non familières avec ces concepts électriques, une approche vulgarisée peut être utilisée [Joh17]. Ainsi, le transistor se comporte comme un interrupteur binaire très rapide (basculement de l'ordre de 10^{-12} seconde).



(a) Schéma d'un transistor NPN [Ger18]

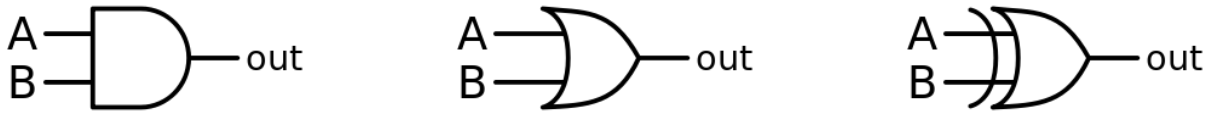


(b) Schéma électrique d'une porte *NON-ET* réalisée à partir de 4 transistors [Wik19a]

FIGURE A.1 – Un transistor est utilisé pour réaliser des portes complexes

A.1.2 Les portes logiques

Les portes logiques sont construites à partir de transistors et permettent l'exécution de différentes instructions ou la capacité de stocker une information (registres). Par exemple, en



(a) La porte `ET` ne laisse passer le courant seulement si les deux entrées sont `VRAI`

(b) La porte `OU` laisse passer le courant si au moins une des deux entrées est `VRAI`

(c) La porte `OU EXCLUSIF` est `VRAI` seulement si les deux entrées ont des valeurs distinctes

FIGURE A.2 – Représentation graphique de trois portes logiques [Wik19d]

associant deux transistors inverseurs en série on construit une porte NON-ET qui laisse passer le courant seulement lorsqu'une entrée a une tension (voir figure A.1b)

Une porte logique possède plusieurs entrées numériques qui peuvent être le résultat d'autres portes logiques (voir figure A.2). Les circuits les plus complexes sont en fait une cascade de milliers de portes logiques comme celles-ci. Il est aussi nécessaire de choisir la signification du passage ou non du courant et construire des portes ayant un « sens ». Il est courant d'utiliser les termes `VRAI` ou *1 logique* lorsque le courant circule et `FAUX` ou *0 logique* pour l'absence de tension.

A.1.3 Algèbre de Boole

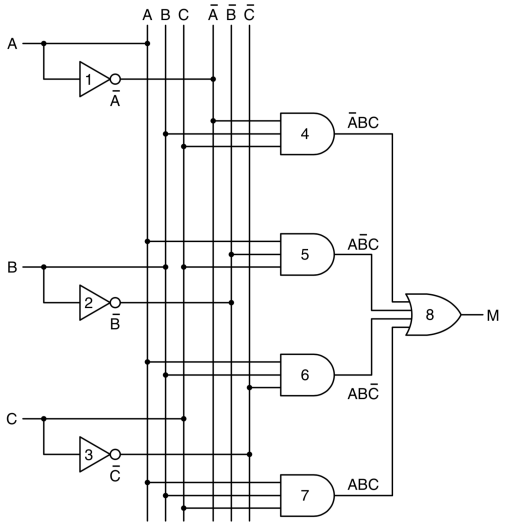
Analyser le fonctionnement de plusieurs portes peut rapidement se complexifier et le recours à des méthodes algébriques est nécessaire. Les portes pouvant avoir des valeurs de 0 ou 1, un nouveau type d'algèbre a été créée : l'algèbre de Boole. Comme pour l'algèbre en base décimale, l'algèbre booléenne utilise des fonctions et des variables pour décrire le comportement d'un système. Les variables utilisées ne peuvent prendre que deux valeurs. Une table de vérité de fonctions générales peut alors être écrite. On peut écrire une table de vérité d'un programme souhaité, qui utilise trois entrées, et déterminer les sorties (M) souhaitées (voir figure A.3a). Grâce à l'algèbre de Boole, on peut convertir cette table en circuit implémentant ce fonctionnement (figure A.3b). L'algèbre de Boole est aussi utilisée pour réduire la complexité d'un circuit sans en changer le comportement, notamment grâce à la fameuse loi de DeMorgan [Hur14] permettant le changement de porte `ET` en porte `OU`. En réduisant la complexité et le nombre de portes, il est possible de réaliser des circuits plus économiques (en utilisant moins de transistors) et plus rapides.

A.1.4 Circuits logiques

À partir des portes logiques et de leur analyse avec l'algèbre de Boole des circuits plus complexes peuvent être élaborés qui peuvent être regroupés en deux grandes familles : les circuits logiques de base et les circuits logiques à mémoire. La principale différence entre les deux vient de leur capacité à retenir une information.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Table de vérité d'un circuit souhaité



(b) Circuit logiques utilisant 8 portes

FIGURE A.3 – À partir d'une table de vérité, on peut générer le circuit logique correspondant [Tan03]

A.1.4.1 Les circuits logiques de base

Les circuits logiques de base contiennent les circuits intégrés qui sont des circuits pouvant comporter quelques centaines de portes (circuit SSI, MSI) ou plusieurs centaines de milliers (circuit LSI et VLSI) [Bar82].

Les circuits combinatoires ne possédant pas de mémoire interne pouvant influer sur le résultat ne dépendent que des entrées. Parmi eux, nous pouvons citer le multiplexeur (qui permet de choisir une entrée parmi plusieurs), les circuits arithmétiques (additionneur, décaleur) et l'horloge.

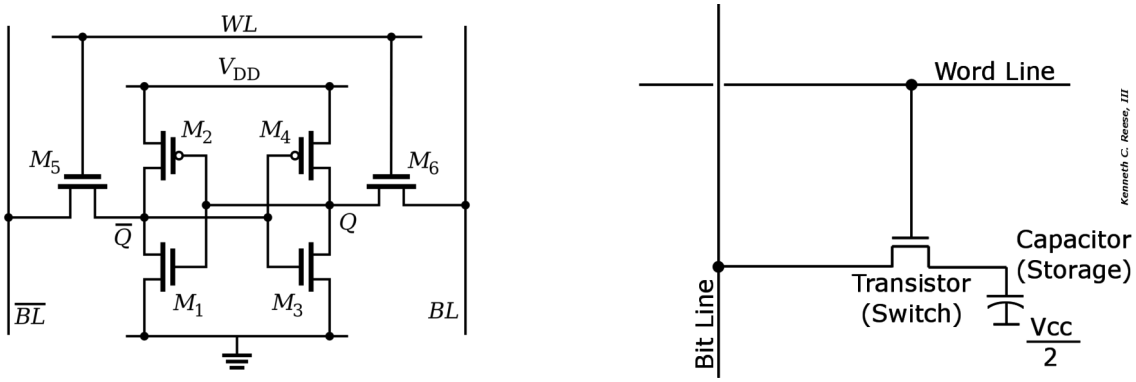
Ces circuits de bases sont utilisés pour construire les unités arithmétiques et logiques des processeurs ou des unité de calcul à virgule flottante (FPU) présentées dans la section A.2.3.4.

A.1.4.2 Les circuits à mémoire

La deuxième famille de circuits comporte un des composants fondamentaux des ordinateurs qui permet la construction de mémoires (figure A.4). Cette capacité de mémorisation est permise grâce à l'utilisation de deux portes `NON-ET` (figure A.4a) ou `NON-OU` (figure A.4b). La particularité de ce circuit, appelé bascule, est la réutilisation de la sortie d'une porte comme entrée d'une seconde, lui permettant le stockage d'une valeur. Les deux entrées d'un tel circuit peuvent être assimilées à une commande de mise à 1 (*set*) et de remise à 0 (*reset*).

(a) Implémentation à partir de portes *NON-ET* (b) Implémentation à partir de portes *NON-OU*

FIGURE A.4 – Réalisation d'une bascule à partir de deux types de portes. La bascule maintient son état actuel tant que les signaux d'entrée ne changent pas.



(a) Mémoire vive statique (SRAM) utilisant 6 transistors (b) Mémoire vive dynamique (DRAM) utilisant un condensateur et un transistor

FIGURE A.5 – Deux types de RAM très utilisés dans les architectures. La différence de complexité de leur circuit électronique explique la différence de prix entre les deux technologies.

A.1.5 Les mémoires RAM

La mémoire à accès aléatoires ou *Random Acces Memory* (RAM) est une mémoire dont le temps d'accès ne dépend pas de la position de l'information contrairement aux disques ou aux bandes magnétiques. Sur ces deux supports, le temps d'accès pouvait varier en fonction de l'emplacement actuel de la tête de lecture et de la prochaine donnée à lire. La RAM est une mémoire volatile, l'information stockée n'est pas persistante lorsque la mémoire n'est plus alimentée. Il y a eu beaucoup d'évolutions des différentes technologies RAM depuis leur création. Il en existe différents types, ayant leurs avantages et leurs inconvénients. Dans une plateforme actuelle, deux types de mémoires RAM sont principalement utilisés : la RAM statique (SRAM, figure A.5a) et la RAM dynamique (DRAM, figure A.5b). La raison principale de la présence de deux types de RAM vient de leur différence de coût de production. La SRAM, bien que plus rapide, est aussi beaucoup plus chère. Cette différence de prix s'explique par l'architecture des deux mémoires (figure A.5).

A.1.5.1 La SRAM

La RAM statique est un circuit logique qui utilise 6 transistors (voir figure A.5a) pour représenter les états 0 et 1 (bien qu'il existe des variantes utilisant 4, 8 ou même 10 transistors). Une SRAM à 6 transistors en utilise 4 pour stocker l'information. Deux transistors additionnels sont utilisés pour contrôler leur accès durant leur lecture ou écriture. Il est courant d'utiliser différents types de SRAM dans les différents niveaux de caches pour optimiser la densité (mémoire de plus grande capacité) ou la vitesse d'accès (latence réduite). Par exemple, le premier niveau de cache est généralement optimisé pour la vitesse d'accès, contrairement aux caches de niveau supérieur de plus grande capacité.

A.1.5.2 La DRAM

La RAM dynamique a une structure plus simple que la SRAM qui n'est composée que d'un transistor et d'un condensateur (voir figure A.5b). La valeur du bit est déterminée par la charge (positive ou négative) du condensateur. Ainsi, que la valeur stockée dans le condensateur soit `VRAI` ou `FAUX` , le condensateur doit être chargé. Cependant, la lecture d'une cellule décharge le condensateur, il faut donc, même lors d'une lecture, le recharger. De plus, à cause des fuites de courant (*power leakage*) les condensateurs doivent être rafraîchis en permanence. La fréquence de rafraîchissement est de l'ordre de 1% à 5% du temps total d'utilisation de la mémoire. Ces deux contraintes font de la DRAM une mémoire très consommatrice en énergie. Grâce à leur faible nombre de transistors, la densité des mémoires DRAM est élevée, permettant la construction de mémoire de grande capacité (en GB). Ainsi, la quasi-totalité des ordinateurs des 50 dernières années ont une mémoire centrale utilisant de la DRAM. La charge et la décharge du condensateur n'étant pas instantanés, la DRAM est beaucoup plus lente que la SRAM (une cellule ne pouvant pas être accédée pendant son rafraîchissement).

A.1.5.3 SRAM vs DRAM

L'avantage de la SRAM est sa rapidité de fonctionnement et sa faible consommation électrique. Contrairement à la DRAM, la SRAM est statique, elle conserve l'information et ne nécessite pas de rafraîchissement périodique pour conserver la donnée enregistrée. Cependant, elle s'efface si aucune tension ne lui est appliquée en continu. Le principal inconvénient de la SRAM vient de son coût de fabrication ainsi que leur faible densité (due à l'utilisation de 6 transistors)

A.1.6 Évolutions des transistors

La vitesse de calcul d'un processeur ou la capacité de stockage d'une mémoire sont directement liées au nombre de transistors disponibles sur une puce. Plus un processeur aura de transistors, au plus il pourra calculer rapidement (ajout de coeur, meilleures unités de calculs). Plus une mémoire aura de transistors, au plus elle pourra contenir de cellules RAM et donc

	SRAM	DRAM
Prix/bit	élevé	bas
Vitesse d'accès	rapide	lent
Latence	0.5-5 ns	50-70 ns
Rafraîchissement	non	oui
Consommation	basse	élevée
Énergie/bit	0,1 pj	1 pj
Densité	faible (6 transistors par bit)	élevée (1 transistor par bit)
Complexité	grande	faible
Utilisation	Cache	Mémoire centrale

Tableau A.1 – Comparaison des caractéristiques principales des mémoires SRAM et DRAM.

avoir une grande capacité de stockage. La performance des systèmes informatiques est donc directement liée aux technologies de transistors utilisées.

A.1.6.1 Évolution du nombre de transistors.

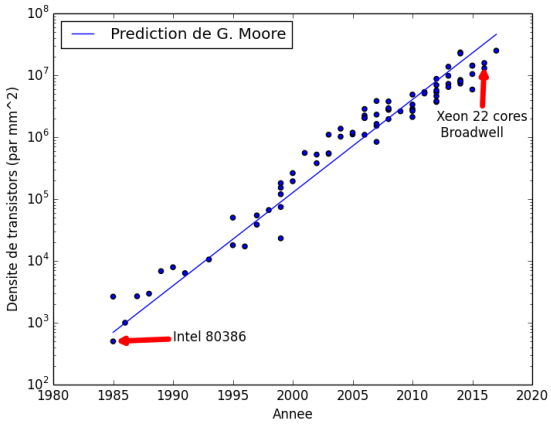
L'évolution du nombre de transistors sur une puce a été prédite par l'un des trois fondateurs de la société Intel, Gordon Moore. En 1965, Gordon Moore prévoit que le nombre de transistors d'une puce doublera chaque année, sur une même surface et pour un coût constant [Moo65]. Il réévaluera cette période à 2 ans en 1975 [Moo75], ce qui correspond parfaitement avec l'évolution réelle jusqu'à ces dernières années (figure A.6a). Les mémoires RAM étant constituées de transistors, leur évolution a aussi été guidée par la loi de Moore (figure A.6b).

A.1.6.2 Coût de la gravure.

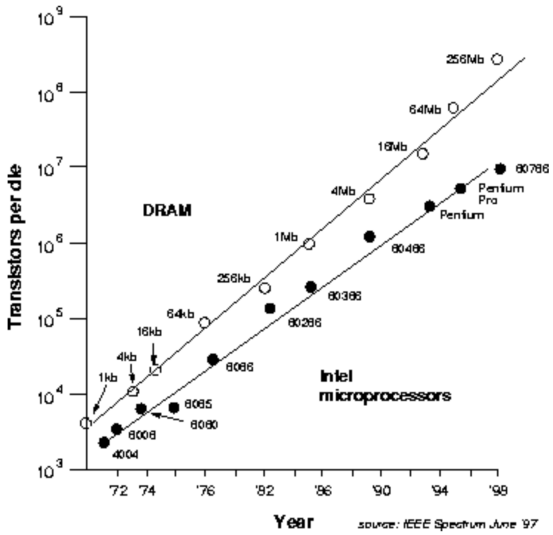
Depuis plus de cinquante ans, les fondeurs (les industriels responsables de la gravure des processeurs) tels que Samsung Electronics, TSMC, Intel et GlobalFoundries ont développé de nombreuses techniques et technologies pour réduire la taille des transistors (voir figure A.7a). Aujourd'hui, de nombreuses étapes sont nécessaires pour transformer une plaque de silicium (la plus pure possible) en processeurs : dopage, déposition d'une couche de résine, gravure, traitement thermique, revêtement par couche mince, découpe, encapsulation... [Ant18].

Les procédés de fabrication étant plus complexes, les usines de fabrication coûtent elles aussi de plus en plus cher. L'augmentation du coût des fonderies a elle aussi été prédite par la seconde loi de Gordon Moore (ou *loi de Rock*), qui estime que leur prix double tous les 4 ans [Sch97].

Bien que les coûts de fabrication augmentent, la taille de gravure s'affine et permet de mettre plus de transistors sur une même surface, permettant ainsi à l'industrie de suivre la



(a) Évolution du nombre de transistors des processeurs Intel (données [Wik19e])



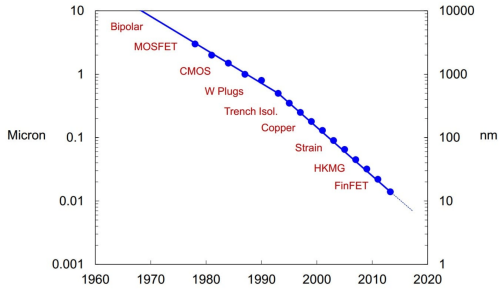
(b) La loi de Moore s'applique aux processeurs comme aux mémoires DRAM [STM05].

FIGURE A.6 – La loi de Moore décrit l'évolution de la densité de transistors qui double tous les deux ans pour un prix constant.

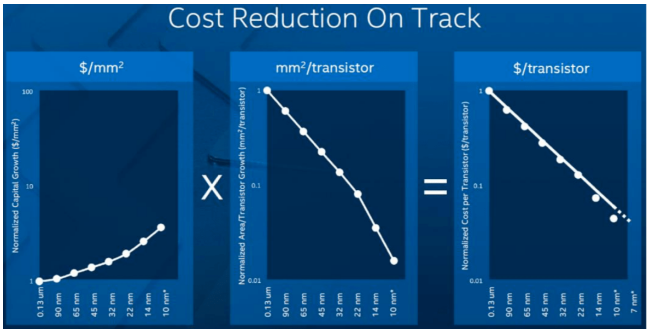
cadence dictée par la loi de Moore (figure A.7b). Cependant, les finesses de gravures utilisées aujourd'hui sont tellement faibles qu'elles atteignent une limite physique, celle de la taille des atomes. À des tailles proches de quelques atomes, les courants électriques ne sont plus stables et la course à la réduction des finesses de gravure n'a jamais été aussi difficile. Voilà plusieurs années que Intel ne parvient plus à descendre sous les 10 nm. Les procédés à mettre en oeuvre pour y parvenir sont si complexes, qu'il est courant de parler de la fin de la loi de Moore [TW17]. En 2019, Samsung annonce qu'il a mis au point une technologie permettant la gravure des premiers processeurs en 3 nm dès 2021 [Adr19].

A.2 L'architecture

Dans la section précédente, nous avons présenté les transistors, les éléments de base des ordinateurs. Ces transistors sont groupés en portes logiques pour construire des circuits électroniques. Cette section présente comment ces circuits sont utilisés pour construire la microarchitecture d'un ordinateur capable d'exécuter les instructions. L'objectif n'est pas de présenter la totalité de l'architecture, mais seulement les éléments importants nécessaires à notre travail de thèse. Nous commençons dans la section A.2.1 par définir une architecture et présenter les principaux modèles utilisés aujourd'hui. Dans les sections suivantes, nous nous intéressons aux évolutions technologiques des processeurs qui ont permis d'améliorer l'efficacité d'exécution des instructions (section A.2.2), d'accélérer leur exécution (section A.2.3) et enfin de rendre leur exécution en parallèle possible (section A.2.4).



(a) Les technologies utilisées pour la gravure ont évolué, rendant les fonderies plus performantes, mais aussi plus chères (source Intel).



(b) Bien que le prix des fonderies augmente, le nombre de transistors gravable sur une puce augmente plus rapidement. Conséquence de la loi de Moore : le prix par transistor diminue exponentiellement.

FIGURE A.7 – Les nouvelles technologies utilisées dans les fonderies permettent de graver des transistors toujours plus petits.

A.2.1 Architecture de processeur : origine et évolutions majeures

Afin d'éviter toute confusion, nous rappelons la définition de la microarchitecture et de l'ISA car ces termes sont souvent confondus dans la littérature :

— **La couche ISA** (*Instruction Set Architecture*) regroupe les instructions, leur mode (système ou utilisateur), les registres utilisables, l'organisation du système mémoire (alignement, espace d'adressage) ... C'est une spécification formelle établie, qui peut être utilisée par plusieurs fabricants de microarchitectures [Tan03]. Intel publie fréquemment la documentation de l'ISA x86 [Int18]. Elle forme l'interface entre le matériel et le logiciel et permet la compatibilité de programme sur des microarchitectures de différents constructeurs. Grâce à la couche ISA, différents langages de programmation peuvent être utilisés pour écrire les applications. Le compilateur s'occupe alors de les traduire dans un langage de bas niveau pouvant utiliser l'ISA (langage assembleur). Ce langage est tellement proche de la couche ISA que les deux termes sont souvent mélangés. Les ISA existantes sont listées dans la section A.2.2.1.

— **La microarchitecture** correspond à l'implémentation matérielle de l'ISA. Ce sont deux couches distinctes, la seconde n'ayant pas forcément besoin d'avoir connaissance de la première (bien que pour des raisons de performances cela soit préférable). En ayant connaissance de la microarchitecture, le compilateur pourra réordonner ou modifier des instructions pour tirer parti du pipeline ou d'un processeur vectoriel. La conception d'une nouvelle microarchitecture doit commencer par choisir l'ISA à implémenter (si possible existante pour permettre la compatibilité des programmes). Les différences principales entre deux microarchitectures implémentant la même ISA sont leur différence de performances et de coût. Les processeurs Intel et AMD implémentent la même ISA `x86`. La performance et le nombre d'instructions supportées par les deux architectures sont

cependant différents.

— Il arrive que le terme d'**architecture** soit employé à la place du terme ISA, par exemple par IBM en 1964 [ABB64]. Aujourd'hui, ce terme est souvent utilisé pour faire référence à la fois à l'ISA et à la *microarchitecture*. Il est courant d'entendre parler d'*architecture x86* pour faire référence à une microarchitecture implémentant l'ISA `x86`.

A.2.1.1 Les premiers processeurs

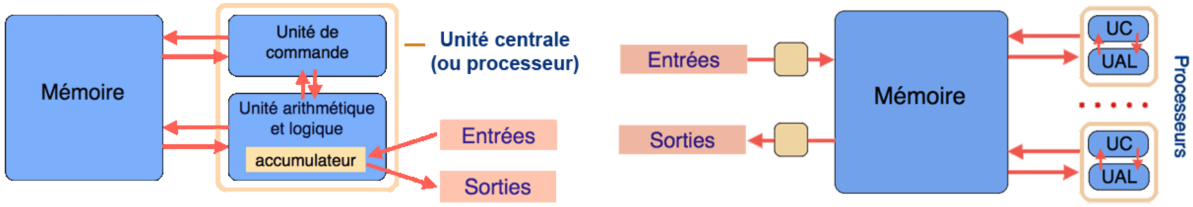
L'origine des premiers calculateurs remonte bien avant l'invention des systèmes électroniques. Les premières machines à calculer sont apparues à la fin du 16e siècle, conçues par Blaise Pascal et Gottfried Wilhelm Leibnitz [Vie96]. Les opérations en base 10 étaient à réaliser par l'utilisateur au moyen de roues dentées. Pour automatiser le recensement des 62 millions d'Américains vivant aux États Unis en 1890, le gouvernement lança un appel d'offres pour construire un système de traitement automatique. Herman Hollerith proposa alors d'utiliser un système de cartes perforées utilisé par des sociétés ferroviaires produit par la société Computing-Tabulating-Recording. C'est en 1924 que la société fut renommée International Business Machines (IBM). De 1937 à 1943, IBM construit pour l'Université de Harvard un calculateur géant capable de multiplier deux nombres de 23 chiffres en six secondes appelé Automatic Sequence Controlled Calculator (ASCC) [Cor16]. Cet ordinateur fonctionnait à partir de dispositifs mécaniques et électromécaniques issus des machines à cartes perforées.

Un autre projet de recherche à l'origine de l'ordinateur est celui mené par Alan Turing durant la Seconde Guerre mondiale, utilisé pour décoder les messages chiffrés d'Enigma de l'armée allemande. Le câblage de ces premiers automates devait être refait lorsqu'on voulait appliquer un changement dans le traitement des opérations. On parlait alors de programmes extérieurs. Le calculateur d'Alan Turing avait une seule utilité, non des moindres, celle de décrypter des messages. Il ne pouvait cependant pas être utilisé pour réaliser d'autres calculs.

A.2.1.2 L'architecture Von Neumann

Bien que la microarchitecture des processeurs ait beaucoup évolué depuis leur création, l'organisation globale n'a pas changé. Aujourd'hui la majorité d'entre eux sont basés sur une architecture qui porte le nom du scientifique à l'origine des premiers schémas : l'architecture *Von Neumann*.

C'est en 1945 que cette architecture a été présentée pour la première fois par John von Neumann dans un papier qu'il n'aura pas le temps de finir *First Draft of a Report on the EDVAC*. Cependant, sur ce papier ne sont pas mentionnés les deux autres contributeurs que sont J. Presper Eckert et John Mauchly. Malheureusement pour eux, l'histoire ne retiendra que le nom du premier. Leur papier décrit les 3 premiers principes qui sont à l'origine des ordinateurs actuels. Le premier voulait que ces machines soient universelles, c'est-à-dire qu'elles puissent exécuter différents types de calculs. Le deuxième concernait leur programmation à l'aide d'instructions organisées dans des programmes qui, comme les résultats intermédiaires,



(a) Architecture Von Neuman initiale (b) Quelques évolutions de l'architecture

FIGURE A.8 – Architecture Von Neumann [1]

peuvent être sauvés en mémoire. On parle alors de programmes enregistrés. Le troisième principe est l'implémentation de la rupture de séquence. Lors de l'exécution d'un programme, l'automate décide des instructions à exécuter, réalise des tests et des comparaisons pour faire des sauts dans le programme. La principale idée de cette architecture est la présence de trois modules principaux (voir figure A.8a) :

— Une unité de traitement (CPU) qui contient une unité arithmétique et logique (ALU) qui s'occupe d'exécuter les opérations de base (addition, multiplication, etc.) ainsi que de registres pour mémoriser les données utilisées pour leur exécution.
— L'unité de contrôle qui lit les instructions et organise leur exécution en s'occupant de demander les données nécessaires à la mémoire.
— Une mémoire qui contient toutes les données, mais aussi le code à exécuter contrairement aux anciennes architectures qui lisaient les programmes sur des cartes perforées, rubans, ou tableaux de connexion.

Bien que l'implémentation de cette architecture ait évolué avec l'apparition de nouveaux matériaux, les architectures modernes des processeurs sont toujours construites sur ce modèle. Deux évolutions de l'architecture initiale ont cependant été appliquées (figure A.8b). Les entrées et sorties ne sont plus gérées directement par l'unité centrale, mais par des microprocesseurs dédiés. L'unité centrale de traitement n'est plus unique depuis l'apparition des processeurs multicoeurs. Ces évolutions ne remettent pas en cause les principes de base énoncés par Von Neumann.

A.2.1.3 Architecture Harvard

Dans leur papier [Neu93], Von Neuman et al. précisent que la façon dont sont stockées les instructions et les données en mémoire doit être la même. Cette configuration la différencie de sa principale concurrente, l'architecture Harvard (figure A.9) qui utilise deux bus pour accéder à deux mémoires réservées : l'une pour les instructions, l'autre pour les données. Cette configuration permet à l'architecture Harvard d'accéder en parallèle aux données et aux instructions. De plus, comme les instructions et les données sont séparées, elles peuvent être stockées sur des supports de différentes performances. On peut ainsi utiliser un support plus cher pour stocker les données avec des mémoires très rapides (SRAM) et stocker les instructions sur des mémoires

1. source : https ://interstices.info/le-modele-darchitecture-de-von-neumann

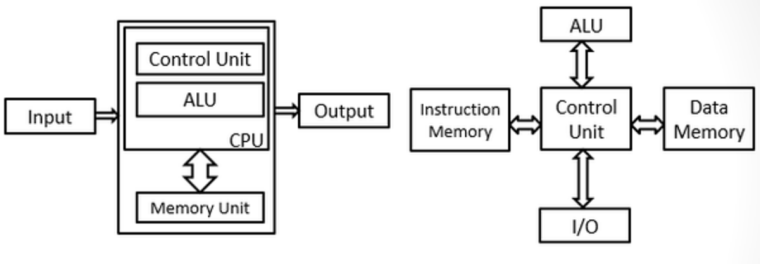


FIGURE A.9 – Les deux principales architectures de processeurs : Von Neumann et Harvard.

moins chers de type ROM. De plus, l'architecture Harvard apporte une sécurité en empêchant les processeurs d'exécuter des instructions provenant du stockage réservé aux données. Cependant, la nécessité d'avoir deux bus rend les puces Harvard plus chères et les performances sont souvent moins bonnes. Un code qui aurait beaucoup d'accès mémoire ne pourrait pas profiter de la disponibilité du canal allant à la mémoire des instructions. Dans une architecture Von Neumann à deux canaux, les bus peuvent être utilisés aussi bien pour des instructions que pour les données. Cependant la performance de ce bus mémoire est à l'origine du déséquilibre de performance des architectures modernes. Ainsi, il est courant d'utiliser le terme de *goulot de Von Neumann* pour désigner ce point faible des architectures modernes.

A.2.1.4 Performance d'une architecture

Le développement d'une nouvelle microarchitecture doit prendre en compte plusieurs facteurs qui peuvent impacter son implémentation (vitesse de traitement des instructions, le coût de fabrication, la fiabilité, consommation électrique, taille). Ces différents facteurs font pression sur les architectes de processeurs qui doivent redoubler d'inventivité pour les satisfaire en implémentant des optimisations matérielles.

Les améliorations ont pour but d'améliorer la performance de l'architecture, la plupart du temps de façon transparente pour l'utilisateur. Cependant, un programmeur n'étant pas averti de ces optimisations pourrait écrire des applications inefficaces. Les développeurs d'applications de Calcul Haute Performance (HPC) étant généralement des scientifiques experts dans leur domaine (physique, chimie, mécanique...), il est fréquent de voir des codes peu efficaces. Cette section liste les principales améliorations des microarchitectures qu'il faut connaître et utiliser pour exploiter le maximum des performances disponibles. Différentes techniques ont été implémentées dans les processeurs au fil des années, rendant ces architectures très complexes. Dans le reste de cette section, nous nous intéressons aux trois principaux moyens d'améliorer leur performance :

— Améliorer l'efficacité des instructions (section A.2.2)
— Accélérer l'exécution des instructions : (section A.2.3)
— Exécuter les instructions en parallèle : (section A.2.4)

A.2.2 Améliorer l'efficacité de l'exécution

A.2.2.1 Jeu d'instructions ISA

Le choix de l'ISA à implémenter est le premier choix à réaliser lors du développement d'une nouvelle microarchitecture. L'ISA utilisée à un impact sur la performance, la facilité de programmation et sur la compatibilité des applications existantes. Les jeux d'instructions couramment utilisés sont séparés en deux grandes familles d'ISA : les jeux d'instructions CICS pour *Complex Instruction Set Computing* et les instructions RISC pour *Reduced Instruction Set Computing*. La principale différence entre les deux est la complexité de leurs instructions.

CISC est la première famille d'instructions à avoir été utilisée massivement. Ces instructions sont dites complexes, car une seule instruction peut à elle seule demander plusieurs opérations à réaliser. Par exemple, une addition CISC s'occuperait de charger les données depuis la mémoire, d'exécuter l'addition pour ensuite sauver le résultat. À l'origine, beaucoup de codes étaient écrits en assembleur, et ce genre d'instructions permettaient au programmeur d'éviter d'écrire de nombreuses lignes de codes souvent redondantes. De plus, les codes générés en CISC sont plus petits et nécessitent donc moins de mémoire, qui à l'origine manquait énormément.

RISC regroupe les ISA dites *simple*. En 1970, John Cocke, alors ingénieur chez IBM, proposa de réduire le nombre d'instructions CISC [CM90]. Le terme *réduit* fait référence au nombre d'instructions plus petit que celles de CISC, mais aussi pour signifier que le travail à réaliser par une instruction était moindre que pour une instruction CISC. Pour réaliser une multiplication entre deux données, on devra alors explicitement charger la première donnée, puis la deuxième et enfin écrire l'instruction qui correspond à la multiplication. Les instructions étant plus nombreuses, le travail des compilateurs est augmenté, mais souvent l'exécution des codes résultante en est réduite. Le RISC fut une réponse apportée à la lenteur de décodage du CISC. Toutes les instructions font la même taille, les architectures nécessitent donc moins de transistors pour les analyser. Les microarchitectures sont donc moins coûteuses et peuvent atteindre des fréquences plus élevées.

Ces deux familles d'instructions ont toutes deux leurs avantages et leurs inconvénients et les puces actuelles comportent des parties qui exécutent des instructions RISC et d'autres en CISC. L'ISA la plus répandue est le `x86` qui se veut être un jeu d'instruction CISC. Les ISA RISC les plus connus sont les ISA ARM, MIPS (utilisé dans le domaine universitaire pour apprendre le langage assembleur), PA-RISC (Hewlett-Packard) et RISC-V. L'ISA CISC la plus utilisée dans les supercalculateurs aujourd'hui est `x86` (processeurs Intel et AMD).

Extensions vectorielles. Pour tirer parti de la puissance de calculs des processeurs et des unités de calculs vectorielles, les ISA ont reçu de nombreuses extensions (tableau A.2). C'est en 1995 que Sun Microsystem introduit son premier jeu d'instruction vectoriel, le Visual Instruction Set auquel Intel répondra en 1997 avec son processeur Pentium MMX et le jeu d'instructions du même nom. Ainsi ces instructions pouvaient réaliser des opérations sur les jeux de données

Année	Nom	Nb registres	Taille (bit)	Registres	Commentaires
1996	MMX	8	64	MM0	Nombre entiers
1999	SSE	8	128	XMM0	120 instr., simple precisions
2006	SSSE3	8	128	XMM0	300 instr., double précisions
2008	AVX	16	128	XMM0	FMA4, op. a 3 opérandes
2011	AVX2	16	256	YMM0	
2013	AVX512	32	512	ZMM	FMA3

Tableau A.2 – Évolutions principales des instructions SIMD `x86`

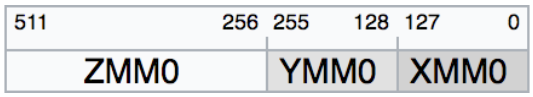


FIGURE A.10 – Découpage d'un registre de 512 bits

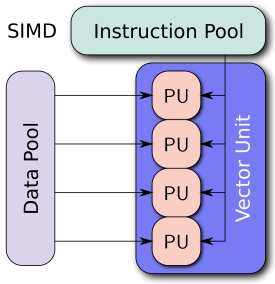
tels que les images ou vidéos de façon très performante. En 1997 AMD viendra améliorer le jeu d'instruction MMX avec l'implémentation des instructions 3DNow ! qui rendait alors possible les opérations vectorielles sur les nombres flottants. La possibilité de faire une opération sur deux nombres flottants par cycle doublait alors la performance des processeurs. Intel répondit alors à cette avancée avec le jeu d'instructions Streaming SIMD Extensions (SSE) en 1999. Il est alors facile d'augmenter la puissance des processeurs : il suffit d'augmenter la taille des vecteurs. L'ISA `x86` a reçu plus de dix extensions dans les vingt dernières années pour s'adapter à l'agrandissement des unités vectorielles. Les principales sont *MMX* (1996), *3DNOW !* (1998), 6 versions de *SSE* de 1999 2008 et enfin *AVX-2* et *AVX-512* en 2013 et 2015. Évidemment, les codes ne peuvent pas profiter de ces instructions sans une microarchitecture capable de les exécuter. Ces microarchitectures sont présentes dans la sous-section suivante.

A.2.2.2 Les processeurs vectoriels

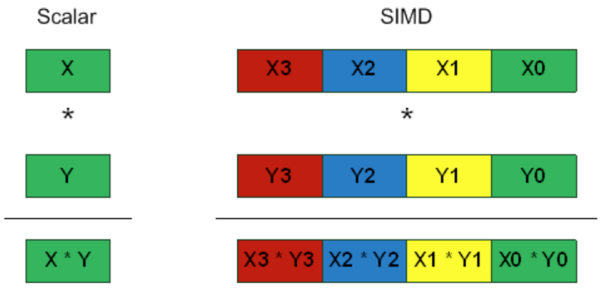
Dans les années 90, les ordinateurs personnels commençaient à être utilisés pour le multimédia (film, image, musique). Dans le but d'augmenter la performance des applications, les architectes ont voulu améliorer l'efficacité des instructions en implémentant des architectures SIMD (Single Instruction Multiple Data de la taxonomie de Flynn présentée dans la section 2.1.3).

L'architecture vectorielle repose sur les mêmes fondements que l'architecture superscalaire (voir section A.2.4.2) : réduire le temps d'exécution d'un code en utilisant le parallélisme et diminuer le coût en transistors de la microarchitecture par la mise en commun du matériel entre plusieurs unités de calculs (figure A.11a). La pression que subit l'unité responsable du *fetch* et du *decode* est donc réduite, car avec une seule instruction vectorielle, le processeur réalise le travail de plusieurs instructions scalaires. Contrairement au processeur scalaire qui exécute une instruction sur une seule donnée, les processeurs vectoriels sont capables d'exécuter

une instruction sur plusieurs données simultanément (figure A.11b).



(a) Un processeur vectoriel exécute une même instruction sur différentes données



(b) Schéma de fonctionnement d'une multiplication vectorielle [2]

FIGURE A.11 – Un processeur vectoriel travaille sur un groupe de données indépendantes. Les opérandes sont alors stockés dans des registres vectoriels capables de charger un vecteur pour y appliquer une même opération.

Ainsi, doubler la largeur des instructions revient à doubler le nombre d'opérations à virgule flottante (FLOP) réalisables par le processeur alors que la consommation électrique augmentera d'un facteur inférieur à 2. Beaucoup d'efforts ont donc été réalisés pour être capables d'utiliser cette technologie. Par exemple la majorité des compilateurs est capable de détecter les zones de codes parallélisables pouvant bénéficier d'instructions SIMD. Cependant, dans la pratique les codes ne sont pas aussi parallèles que nous le souhaiterions et certains aspects du code empêchent de tirer la totalité de la performance disponible. En effet, si les instructions sont dépendantes les unes des autres, il est alors impossible de les calculer simultanément rendant la partie vectorielle inutilisable. Aussi, les performances peuvent être réduites si les données accédées ne sont pas continues en mémoire. Cela demande donc un travail supplémentaire pour repenser les structures de données et s'assurer que les données transférées sur le bus mémoire sont des données utiles (voir le concept de ligne de cache dans la section A.3.3).

A.2.3 Accélérer l'exécution des instructions

Utiliser des instructions plus efficaces telles que les instructions vectorielles ou les FMA est un premier moyen d'améliorer la puissance de calcul d'un processeur. Un deuxième moyen est de les exécuter plus rapidement. Cela peut être réalisé en accélérant la fréquence d'un processeur, ou en utilisant des matériels spécialisés pour l'exécution de ces instructions.

A.2.3.1 Lien entre fréquence et performance

Les processeurs sont des circuits électroniques dits *synchrones*. Leur fonctionnement est cadencé par une horloge donnant des impulsions régulières aux composants du circuit pour organiser leur synchronisation. Lorsqu'une instruction est exécutée par le processeur les passages

2. source :https://software.intel.com/en-us/articles/ticker-tape-part-2

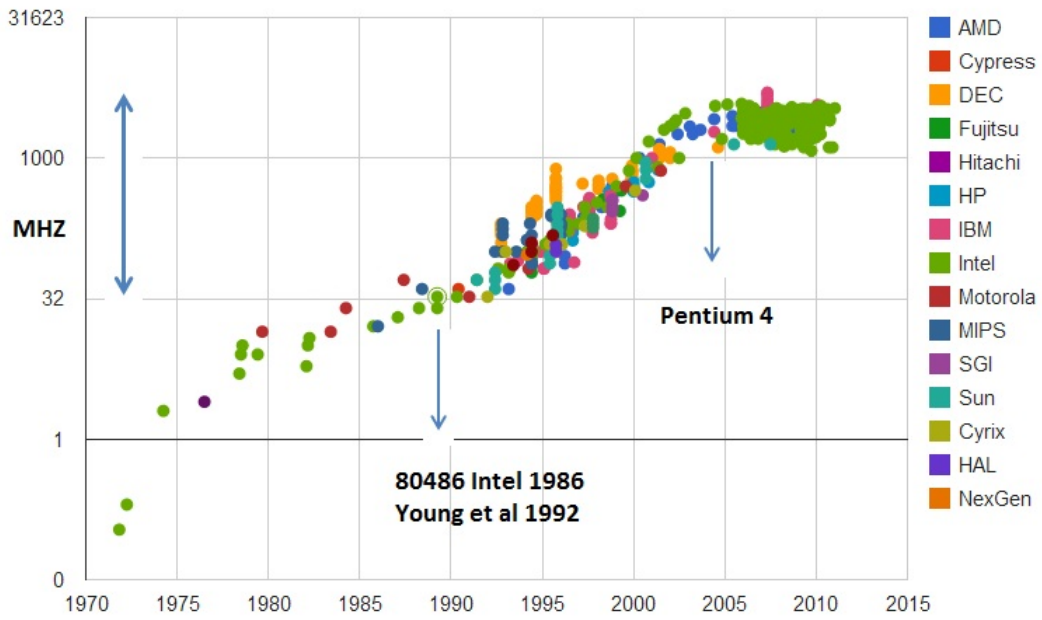


FIGURE A.12 – Évolution de la fréquence des processeurs [4].

dans les différentes étapes de la microarchitecture, notamment du pipeline (voir section A.2.4.1), sont régis par cette horloge. Plus l'horloge est rapide, plus l'exécution du programme l'est aussi. Cependant, la durée séparant deux signaux (un cycle) ne peut pas être réduite à l'infini. En effet, les vitesses d'horloges sont si rapides, qu'il est nécessaire de laisser suffisamment de temps au signal électrique de se propager dans le circuit. La taille du processeur et sa fréquence sont donc liées, c'est notamment pour cette raison qu'il n'existe pas de processeur mesurant plusieurs dizaines de centimètres. À une fréquence de 3 GHz, le signal électrique, qui se déplace à la vitesse de la lumière, ne peut parcourir que quinze centimètres entre deux cycles. En plus du temps de propagation, il faut aussi prévoir le temps de préparer et recevoir la communication. L'augmentation de la fréquence des processeurs a donc une première limite physique infranchissable bien que d'autres limites présentées dans cette partie soient encore plus contraignantes.

La fréquence des processeurs a beaucoup évolué depuis les premiers processeurs (figure A.12). Les premiers processeurs *Pentium* d'Intel utilisaient des fréquences de 60 MHz en 1993. Pendant plus de dix ans, les fréquences ont évolué chaque année d'un facteur de 40% tous les ans pour atteindre des vitesses de plusieurs Gigahertz. En 2000, Intel annonçait la fabrication de processeurs cadencés à 10 GHz [3] pour les années 2005. Pourtant, en 2019, nous sommes encore loin d'utiliser des processeurs avec de telles cadences. Mis à part les systèmes surcadencés, utilisant des systèmes de refroidissement liquide, il est rare de voir des systèmes utiliser des processeurs récents à plus de 5 GHz. .

3. source https://www.clubic.com/actualite-1791-des-processeurs-intel-10-ghz-pour-2005.html

Le but de cette section est de comprendre pourquoi l'évolution de la fréquence des processeurs s'est arrêtée brusquement autour des 4 GHz. Pour cela, nous expliquons comment la loi de Moore a permis d'en arriver là, et quelles sont les limites physiques qui empêchent de poursuivre cette évolution.

A.2.3.2 Loi de Dennard

Comme constaté dans la section précédente, la fréquence de l'horloge d'un processeur est limitée par le temps nécessaire au signal de se propager entre deux éléments d'un circuit. Plus les transistors seront proches, plus la vitesse de propagation sera faible. La loi de Moore prévoit que le nombre de transistors double tous les deux ans pour une surface donnée. Cela est rendu possible par l'affinement de la gravure permettant de réaliser des transistors toujours plus petits et donc plus proches. Avec plus de transistors, les processeurs peuvent utiliser des pipelines plus complexes. Ainsi, en divisant les étapes les plus longues les processeurs sont capables d'utiliser des fréquences plus élevées. Nous expliquons comment l'évolution de la fréquence des processeurs est intimement liée avec la loi de Moore en utilisant la formule de la consommation électrique d'un circuit CMOS [MAR14] :

$$P = QfCV^2 + VI_{leakage} \tag{A.1}$$

L'intérêt de cette formule est d'apprécier comment la puissance et la fréquence d'un processeur est impactée par la loi de Moore. Pour cela, nous étudions sa variation entre deux générations de processeurs, c'est-à-dire une période de deux ans :

- **Le nombre de transistors Q :** D'après la loi de Moore, le nombre de transistors double tous les deux ans pour deux circuits de même surface. Ainsi la surface des transistors est divisée par deux. Leur longueur et leur largeur sont ainsi réduites d'un facteur $\sqrt{2}$. Cette valeur est appelée facteur de *scaling* et a une valeur de 1.44.
- **La capacité du circuit C :** La capacité d'un transistor peut être calculée par la formule suivante $C = \frac{S \times e}{d}$. Avec S la surface du transistor, e la permittivité électrique du matériau utilisé (pouvant être considéré comme fixe entre deux générations), et d la distance séparant la grille et le semi-conducteur (isolant). Comme vu précédemment, la surface S d'un transistor est divisée par 2 entre deux générations. La distance d est-elle réduite par un facteur $\sqrt{2}$. Tous les deux ans, la capacité d'un transistor est donc réduite d'un facteur $\sqrt{2}$.
- **La fréquence du circuit f :** La fréquence d'utilisation d'un transistor dépend essentiellement de la vitesse à laquelle la grille peut être chargée ou déchargée. Diminuer sa capacité diminue du même facteur ce temps de remplissage. Entre deux générations, la fréquence est supposée augmenter d'un facteur $\sqrt{2}$. Cette valeur correspond bien à l'augmentation de 40% constatée dans l'introduction de cette section.
- **La tension de fonctionnement V :** La tension est proportionnelle à la finesse de grave utilisée. Diviser la finesse de gravure par un coefficient de $\sqrt{2}$ revient à diviser la tension de fonctionnement par le même facteur.

4. source : https://en.wikipedia.org/wiki/Beyond_CMOS

Paramètre	Coefficient multiplicateur (tous les deux ans)
Finesse de gravure	$\frac{1}{\sqrt{2}}$
Nombre de transistors par unité de surface	2
Tension d'alimentation	$\frac{1}{\sqrt{2}}$
Capacité d'un transistor	$\frac{1}{\sqrt{2}}$
Fréquence	$\sqrt{2}$

Tableau A.3 – Résumé des impacts de la diminution de la finesse de gravure d'un facteur $\sqrt{2}$ [5].

— **Les courants de fuites $I_{leakage}$:** Les courants de fuites sont considérés comme négligeables (pour le moment).

Variation de P. À une tension de fonctionnement V égale entre deux générations de processeurs et en reprenant les variations des différentes valeurs, nous constatons que P ne varie pas entre deux générations. Le tableau A.3 résume les différents impacts que la diminution de la finesse de gravure a sur les différentes propriétés d'un circuit. On remarque deux choses concernant l'évolution de P entre deux générations de circuit. La première est que l'augmentation de la fréquence est compensée par la baisse de la capacité du circuit. La deuxième est que la baisse de tension compense le nombre de transistors. Ainsi, la consommation électrique d'un processeur ne varie pas entre deux générations, bien que la fréquence augmente de 40% et que le nombre de transistors soit doublé. Cette propriété est connue sous le nom de Loi de Dennard [Den+74] qui assurait en 1974 que la densité énergétique resterait constante entre deux générations de processeurs. Cette propriété est restée vraie durant 30 ans, permettant l'augmentation de la performance des processeurs sans augmenter drastiquement leur consommation électrique.

A.2.3.3 Fin de Dennard

Les équations de Dennard se sont appliquées pendant plus de 30 ans, voyant la fréquence des processeurs augmenter de 40% tous les 2 ans (voir figure A.13). Cependant, l'équation A.1 ignorait les courants de fuite $I_{leakage}$ alors peu significatifs. Avec la miniaturisation des transistors, ces fuites augmentent exponentiellement pour des tailles de gravure inférieures à 65nm [MAR14]. Alors que la loi de Dennard prévoyait une consommation électrique constante entre deux générations de processeur, ces fuites de courant vont faire augmenter la consommation des puces d'un facteur 2. Ceci entraîne une forte évolution de la densité électrique à chaque nouvelle génération (voir figure A.14) impactant la consommation électrique des processeurs (voir figure A.13). Plus la fréquence des processeurs est élevée, plus les transistors sont activés

5. source : https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/La_consommation_d%27%C3%A9nergie_d%27un_ordinateur

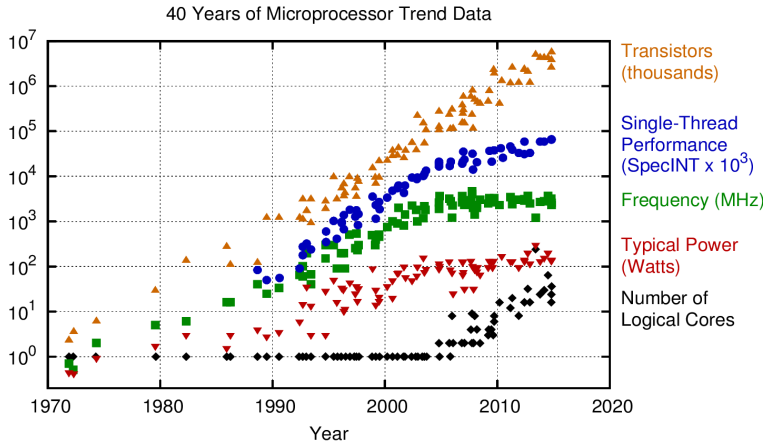


FIGURE A.13 – Évolution des caractéristiques des processeurs [Rup15].

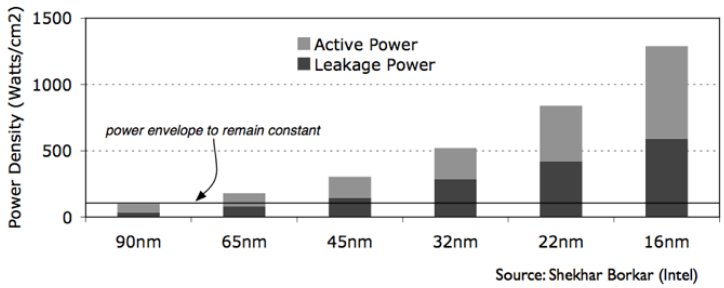


FIGURE A.14 – La miniaturisation de l'isolant nécessaire au fonctionnement des transistors permet le passage de courant de *fuite*.

augmentant d'autant les courants de *fuite*. La chaleur dégagée par les puces n'est alors plus soutenable et l'évolution de la fréquence des processeurs s'arrête ainsi autour des années 2005.

Bien que la fréquence des processeurs n'augmente plus depuis plus de 10 ans, la figure A.13 montre que la performance des processeurs continue bien d'augmenter. En effet, si la loi de Dennard n'est plus valide, la loi de Moore l'est encore après 2005. Les processeurs reçoivent toujours plus de transistors permettant d'augmenter la complexité des pipelines. Ces transistors sont alors utilisés pour construire des processeurs avec plusieurs coeurs. Présentés dans la section A.2.4.3, il est intéressant de remarquer que l'augmentation du nombre de coeurs commence exactement quand la fréquence n'augmente plus. Les processeurs sont alors capables de réguler la chaleur d'un processeur en plaçant les processus peu actifs sur les points chauds de la puce. Pour limiter la consommation électrique, des techniques de management d'alimentation sont alors mises en place pour éteindre certaines parties du processeur inutilisées. L'utilisation de différentes fréquences est aussi largement utilisée. Les processeurs possèdent des fréquences dites *turbo*, leur permettant d'atteindre des fréquences très élevées pendant un court laps de temps ou lorsque le processeur n'est pas pleinement utilisé (coeurs et calculs vectoriels inutilisés).

Année	Architecture Intel / AMD		Simpe p.	Double p.	Opérations
2008	Nehalem	K10	8	4	4 add. et 4 mul.
2011	Sandy Bridge	Bulldozer	16	8	8 add. et 8 mul.
2013	Haswell & Skylake		32	16	8 FMA (mult. + add.)
2016	Xeon Phi KNL		64	32	8 FMA (mult. + add.)

Tableau A.4 – Evolution de la performance des FPU

A.2.3.4 FPU

Le second moyen d'accélérer l'exécution des instructions est d'utiliser des matériels spécialisés pour leur exécution. L'unité de calcul en virgule flottante (FPU) est un composant du processeur permettant de réaliser les opérations sur les nombres à virgule flottante. À l'origine ce module était séparé du processeur et il convenait à l'utilisateur de choisir s'il voulait ou non en brancher une sur la carte mère dans l'emplacement qui lui était alors dédié. Pour des questions de coûts d'intégration et de performance, la FPU est depuis 1989, avec la sortie du processeur Intel 80486, intégrée au processeur.

Au fil des années, la complexité de la FPU a augmenté. Quand elle n'était capable d'exécuter que de simples opérations à l'origine, elle peut désormais réaliser des opérations complexes (division, racine carrée, exponentielles ou des fonctions trigonométriques). De plus, des instructions fusionnées ont fait leur apparition en 2013 dans les processeurs *Haswell* de Intel et Piledriver pour AMD. Elles ont la particularité de réaliser 2 opérations en un seul cycle d'horloge du processeur. Connues sous le nom de FMA pour *fused multiply-add* elles sont capables d'exécuter l'instruction $a \leftarrow b * c + d$ en un seul cycle.

Enfin, les unités de calcul modernes sont capables d'exécuter une opération sur plusieurs données à la fois. Ce type d'instructions est dit *vectoriel*. Ces instructions sont performantes pour les algorithmes qui doivent exécuter une même opération sur plusieurs données (un vecteur par exemple). Les différentes évolutions de la FPU sont responsables de la forte augmentation de la puissance de processeur. En effet, usuellement la puissance d'un processeur est donnée par le nombre de calculs flottants qu'il peut exécuter par cycle. Le tableau A.4 montre comment le nombre de FLOP par cycle évolue : pour Intel, cette performance a été multipliée par deux à chaque nouvelle version de l'architecture et les FPU modernes exécutent 8 fois plus de calculs qu'en 2008.

A.2.3.5 Optimisations

Dans les sous-parties précédentes nous avons présenté comment l'évolution de la fréquence et l'utilisation de matériel spécialisé comme une FPU permet d'accélérer l'exécution d'une instruction. Pour différentes raisons (dépendance, manque d'un opérande), les instructions ne peuvent pas être exécutées à la vitesse maximale théorique prévue par le processeur. Pour maxi-

miser le nombre d'instructions exécutées chaque cycle, les processeurs ont reçu de nombreuses optimisations.

Exécution dans le désordre. Le but de cette optimisation est de cacher l'attente de données du processeur de la mémoire. Cette avancée est apparue sur les processeurs Intel en 1995 avec le *Pentium P6*. Le principe de l'exécution dans le désordre est d'exécuter les instructions dans un ordre différent que celui donné par le code source. Ainsi lorsqu'une instruction doit attendre une donnée, au lieu de perdre plusieurs cycles à attendre ces données, le processeur va exécuter les instructions qui suivent et finira d'exécuter la première quand la donnée sera chargée dans un registre. Cependant, ne pas exécuter le programme dans l'ordre initial peut fausser les résultats. C'est alors au processeur de s'assurer que les instructions permutées ne sont pas dépendantes. Il existe trois types de dépendances.

— La première est une *lecture après écriture* (Read After Write ou RAW) : une instruction lit une donnée écrite par une instruction la précédant.
— La deuxième est une *écriture après lecture* (WAR) : une première instruction lit une donnée qui est modifiée par une instruction la suivant.
— La dernière dépendance est une *écriture après écriture* (WAW) : deux instructions écrivent sur la même donnée.

Pour éliminer ces deux dernières dépendances, les processeurs possèdent plus de registres que ceux adressables par le programme. Il les utilise pour éliminer les dépendances WAR et WAW grâce à des techniques de renommage de registres [Wan+01]. L'extrait A.1 donne un exemple pour chaque type de dépendances. Dans chaque cas, la valeur de B n'est pas la même si les deux assignations ne sont pas exécutées dans le même ordre, suivant cet ordre, B peut valoir 10 ou 15. Pour ce faire, le processeur possède une fenêtre de plusieurs instructions, aussi appelée (à tort) l'*exécution queue*. En effet, cette liste n'a pas vocation à être exécutée dans l'ordre, c'est un rassemblement d'instructions qui ont des dépendances entre elles. Le processeur vient mettre à jour cette liste pour essayer d'exécuter des instructions qui n'ont plus de dépendances avec une donnée ou une autre instruction. Pour pouvoir bénéficier de l'exécution dans le désordre, le processeur doit donc détecter si les instructions sont dépendantes pour pouvoir les réordonner. Aussi, le programmeur peut aider le processeur dans son travail en évitant au maximum les dépendances entre les instructions. La complexité apportée par le système d'exécution dans le désordre peut être la source d'attaque comme la récente faille *Meltdown* des processeurs Intel [Lip+18].

Prédiction de branchement L'exécution dans le désordre fonctionne tant que suffisamment d'instructions sont disponibles pour être exécutées. Cependant, lorsqu'un programme contient un branchement conditionnel, le processeur doit attendre que sa condition soit testée. Si cette condition utilise une variable modifiée dans les instructions précédentes, le processeur doit attendre d'avoir le résultat du test pour connaître les futures instructions à exécuter. Pour maximiser l'utilisation du pipeline, le processeur peut essayer de prédire le résultat du test et ainsi continuer l'exécution. S'il s'est trompé sur la prédiction, il doit alors annuler les instruc-

```
//———————— Read after Write ————————
int A, B = 0
A = 5;
B = A + 10;
//Result: B == 10 ou B == 15

//———————— Write after Read ————————
int A, B = 0
B = A + 10;
A = 5;
//Result: B == 10 ou B == 15

//———————— Write after Write ————————
int B = 0
B = 10;
B = 15;
//Result: B == 10 ou B == 15

```

Extrait A.1 – Exemples de dépendances entre deux instructions.

tions déjà exécutées et reprendre l'exécution des autres instructions. Le temps nécessaire pour la reprise de l'exécution après une mauvaise prédiction dépend donc de la taille du *pipeline* utilisé (plusieurs dizaines de cycles). La prédiction de branchement peut être implémentée par deux méthodes. Le processeur peut posséder un matériel spécifique, appelé prédicteur de branchement (*branch predictor*), qui utilise des méthodes de statistiques. Lorsqu'un branchement est faux plusieurs fois de suite, il peut estimer qu'il y a une grande probabilité qu'il le soit aussi à l'itération suivante et éviter d'attendre le résultat du test pour continuer l'exécution. Le processeur peut aussi, à partir de l'adresse de destination, comprendre si la condition et le saut sont utilisés dans une boucle ou si c'est un retour de fonction. Une mauvaise prédiction pouvant fortement impacter l'exécution, les processeurs implémentent des prédicteurs de branchement toujours plus complexes. Ce matériel représente une part non négligeable du processeur, et des travaux ont pour objectif d'en comprendre leur fonctionnement [MMK02]. La deuxième façon d'implémenter la prédiction est réalisée statiquement, par le compilateur. À la lecture du code, le compilateur peut deviner qu'une boucle ne verra son branchement vrai qu'a la fin de son parcours. Il peut alors calculer le nombre d'itérations à réaliser et éviter le test à chaque itération. Comme pour le mécanisme d'exécution dans le désordre, la complexité apportée par le prédicteur de branchement a donné lieu à une importante faille de sécurité découverte en 2018 par les chercheurs de Google appelée Spectre [Koc+18].

A.2.4 Exécuter les instructions en parallèles

La loi de Moore a assuré aux processeurs un gain constant de transistors chaque année. Ils peuvent être utilisés pour implémenter de nouvelles fonctionnalités matérielles permettant d'exécuter les instructions en parallèle pour accélérer les applications. Les processeurs ont reçu

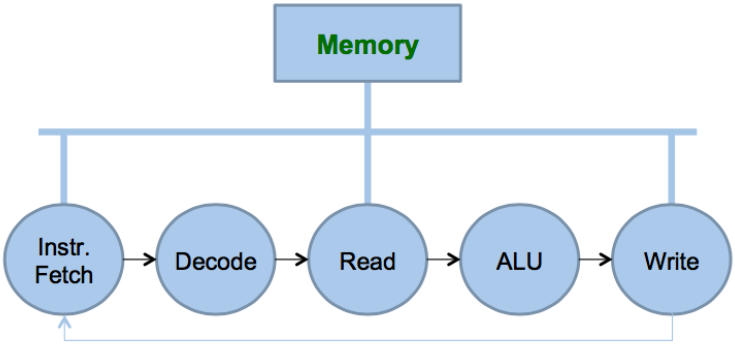


FIGURE A.15 – Représentation simplifiée d'un pipeline de 5 étapes.

de nombreuses améliorations dont les principales sont présentées dans cette section :

— Le pipeline
— Les processeurs superscalaires
— Les coeurs

A.2.4.1 Le pipeline

L'utilisation d'instructions CISC toujours plus complexes a pour effet d'allonger le temps nécessaire à leur exécution qui dure alors plusieurs cycles. Les instructions complexes nécessitent plusieurs opérations : le chargement depuis la mémoire (*fetch*), le décodage (*decode*), le chargement des données nécessaires (*memory*), l'exécution (*execute*) et l'enregistrement du résultat (*write-back*). Pendant ces différentes étapes, la totalité de l'unité d'exécution n'est pas disponible pour les instructions suivantes.

La chaîne de traitement du processeur, ou *pipeline*, est une implémentation matérielle d'un module qui permet de découper l'exécution d'une instruction en plusieurs étapes (figure A.15). Cette technique peut être vu comme une analogie à l'utilisation de chaîne de montage. Datant de plus d'un siècle, la technique de la chaîne de montage a été abondamment utilisée par des industriels tels que Louis Renault et Henry Ford [Wol57].

Implémentation En informatique, la technique de *pipeline* est utilisée pour exploiter le parallélisme d'instructions (ILP) (figure A.16). Il est commun de présenter la notion de pipeline avec un pipeline de 5 niveaux :

— **Recherche de l'instruction** ou *fetch* : cette première étape charge l'instruction à exécuter depuis la mémoire principale dans un registre du processeur. Grâce à un compteur interne, le registre *Program Counter*, le processeur connaît l'adresse mémoire de la prochaine instruction à charger. Pour améliorer le temps d'accès aux instructions, le processeur possède un tampon (*instruction buffer*) contenant plusieurs instructions d'avance.

(a) Processeur sans pipeline

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

(b) Processeur avec un pipeline à 5 étages

FIGURE A.16 – Pipeline[6] : en séquençant les instructions, le processeur est capable d'exécuter des étapes différentes en parallèle (*IF : instruction fetch, ID : instruction decode, EX : execution, MEM : memory, WB : write back*). Le nombre de cycles nécessaires pour l'exécution de 3 instructions passe alors de 15 à 9 cycles

— **Décodage** ou *décode* : une fois que l'instruction est chargée, elle est décodée pour déterminer l'action à exécuter et les données nécessaires.
— **Execution** ou *execute* : en fonction du décodage réalisé, l'instruction est exécutée : utiliser l'ALU pour faire une opération ou calculer une adresse.
— **Accès mémoire** ou *memory* : réalise un accès mémoire (*load* ou *store*) lorsqu'une instruction le nécessite.
— **Ecriture du résultat** ou *write back* : enfin le processeur doit enregistrer le résultat produit par l'étape *execute*. Si c'est un branchement, il modifie le registre *Program Counter* (*branch*). Si c'est une opération arithmétique, il sauvegarde le résultat dans l'adresse destinataire décodée par la deuxième étape.

Le fait de partager l'exécution d'une instruction en sous-étapes permet de commencer l'exécution de la suivante pendant que l'instruction actuelle est encore dans la chaîne d'exécution (voir figure A.16). Son utilisation ne réduit pas le temps d'exécution d'une instruction (5 cycles sur la figure A.16a). Celles-ci doivent tout de même passer une à une par chaque étape de la chaîne. Le *pipeline* permet d'améliorer la cadence d'exécution en maximisant l'utilisation de chaque ressource à un même moment (figure A.16b). On peut par exemple commencer à charger la prochaine instruction (étape *fetch*), alors que l'instruction actuelle est en train d'être exécutée (étape *execute*). Sur la figure A.16b on assiste à l'exécution de 5 instructions, au premier temps une seule instruction est exécutée, à l'étape IF pour *instruction fetch*. Ensuite, une nouvelle instruction est chargée (opération IF) pendant que la première est passée à l'étape suivante (opération ID). Ainsi au bout de 5 cycles, chaque étape du pipeline est utilisée (partie en verte).

Taille du pipeline. En 1939 IBM conçoit le premier processeur avec pipeline. Ce n'est qu'en 1989 qu'Intel produira le sien (Intel 80486). Le nombre d'étapes (ou profondeur du pipeline)

6. source - https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs)

était de deux à l'origine. Cette taille a augmenté au fil du temps atteignant 31 étapes pour l'architecture du Pentium 4 Prescott d'Intel en 2004.

Complexité de la gestion du pipeline. L'utilisation d'un *pipeline* n'est pas toujours optimale et plusieurs facteurs peuvent affecter sa performance. Le principe du pipeline repose sur le concept de commencer à exécuter des instructions avant que la précédente ne soit terminée. Cela peut être rendu impossible par la dépendance entre deux instructions et par l'utilisation de branchements conditionnels [ED87]. Lors de l'évaluation d'un tel branchement, le pipeline ne peut pas commencer à exécuter les instructions suivantes sans connaître son résultat. Le processeur doit alors attendre (*stall*) plusieurs cycles avant de continuer. Une optimisation de prédiction de branchement a été implémentée pour éviter ces états de *stall* (voir section A.2.3.5). De plus, pour permettre la bonne utilisation du pipeline, des mémoires tampons doivent être disposées entre chaque étape pour mémoriser les différents résultats intermédiaires. Lorsque le processeur exécute plusieurs processus, il doit veiller à terminer l'exécution des instructions avant de commencer celles du processus suivant. La complexité de sa gestion le rend vulnérable aux attaques informatiques (voir section A.2.3.5).

A.2.4.2 Processeurs superscalaires

Le pipeline apporte un niveau de parallélisation horizontale. Les processeurs ont reçu une autre amélioration apportant au pipeline une parallélisation verticale.

Principe Un processeur est dit *superscalaire* s'il est capable d'exécuter plus d'une instruction simultanément. Le nombre d'instructions par cycle d'horloge (IPC) peut alors être supérieur à 1. Le principe est d'implémenter un second pipeline (ou plus) capable d'exécuter les instructions telles que sur la figure A.16b. Intel proposa son premier processeur superscalaire en 1989 avec le processeur Intel 80486. Il possédait un pipeline à cinq étages proche de celui présenté dans la section précédente. Pour pouvoir l'utiliser, le processeur doit déterminer si deux instructions peuvent être exécutées en parallèle (sans dépendance et utilisant des ressources matérielles différentes). La figure A.17 montre comment une implémentation superscalaire du pipeline fonctionne.

Implémentation Il existe deux façons de transformer un processeur scalaire en superscalaire. La première est de dupliquer matériellement chaque étape pour obtenir deux pipelines distincts (processeurs *superpipeline*). On peut citer le processeur Intel Pentium dont la totalité du pipeline n'est pas dupliquée. Le processeur a une fenêtre de plusieurs instructions prêtes à être exécutées. Seule la phase d'exécution est dupliquée. Il possédait deux unités d'exécution u et v qui pouvait exécuter des instructions de types différents (opérations flottantes ou entières) en parallèle. Le deuxième moyen d'implémenter le parallélisme d'instruction d'un processeur superscalaire repose sur le fait qu'une étape peut nécessiter moins d'un demi-cycle d'horloge

7. source : https://fr.wikipedia.org/wiki/Processeur_superscalaire

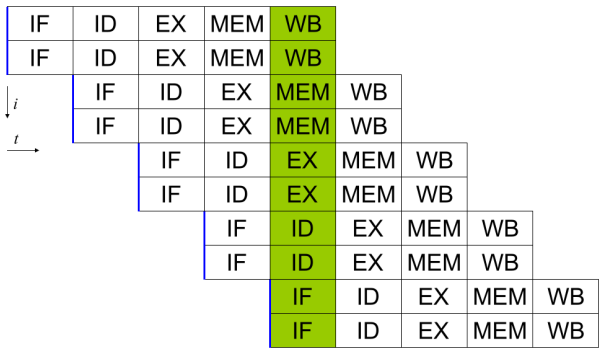


FIGURE A.17 – Fonctionnement d'un processeur superscalaire possédant deux pipelines. [7].

pour être exécutée. Une étape (ou micro-instruction) du pipeline peut donc s'occuper de deux instructions différentes pendant un cycle d'horloge, en utilisant sa propre horloge interne. Cette méthode a le bénéfice de ne pas avoir à dupliquer le pipeline matériellement.

Les principales limitations à l'implémentation d'un pipeline sont les dépendances et les conflits. Cela peut être une dépendance entre les données de plusieurs instructions ou un conflit d'accès à une même ressource (ALU, FPU). Le processeur est alors en charge d'orchestrer les différentes instructions pour rendre possible le parallélisme en utilisant des stratégies *d'émission* [Joh89] des instructions. Cette stratégie doit veiller à conserver la validité du programme en veillant à l'ordre de lecture des instructions, de leur exécution et de leur actualisation des registres (ou de la mémoire).

Exemple du Pentium 4 Pour bien comprendre le déroulement de l'exécution du pipeline d'un processeur superscalaire, nous choisissons de détailler le fonctionnement de processeur Intel Pentium 4 [Sta03] donc le schéma de la microarchitecture est présenté sur la figure A.18. Le processeur exécute les *micro-ops* en utilisant un pipeline d'au moins 20 étages en veillant à respecter les dépendances, dont les principales étapes sont décrites ici :

1. Le processeur lit les instructions (CISC) depuis le cache L2 par groupes de 64 octets dans l'ordre du programme pour profiter de l'effet de localité. Bien que la prédiction de branchement puisse modifier cet ordre.
2. Chaque instruction (pouvant être de taille différente) est décodée et traduite en une à quatre instructions RISC de 118 bits (*micro-ops*).
3. Ces micro-ops sont ensuite stockées dans un buffer (*Trace Cache*) permettant l'utilisation de l'exécution dans le désordre (voir section A.2.3.5).
4. Ensuite, le processeur procède au renommage des registres. Il existe 16 registres architecturaux (utilisable par le code), mais 128 registres physiques sont réellement implémentés. Les *micro-ops* peuvent ensuite être stockées dans deux listes d'attente distinctes utilisant une discipline *FIFO*.
5. L'ordonnanceur choisit ensuite dans les deux files les instructions qui possèdent leurs opérandes et dont l'exécution peut être réalisée. Suivant le type d'instruction, elles sont

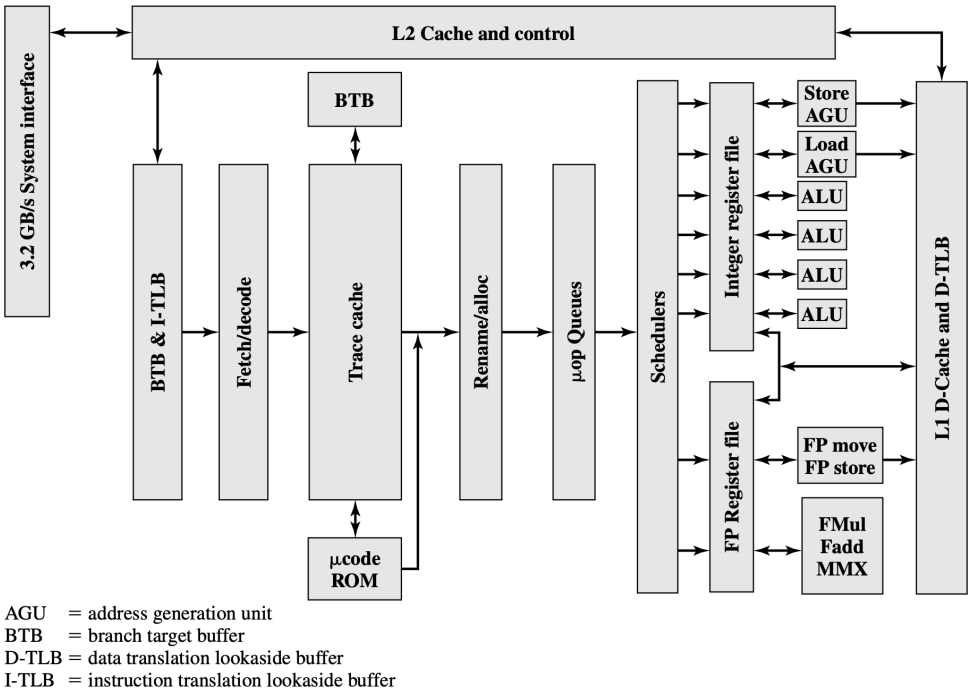


FIGURE A.18 – Diagramme en bloc du Pentium 4 [Sta03]

envoyées (jusqu'à 6 à la fois) vers l'unité d'exécution correspondante (calcul entier ou flottant) en utilisant les différents ports.

Les architectures actuelles ont beaucoup évolué depuis le premier pentium. Le détail de la microarchitecture Skylake d'Intel peut être consulté sur la figure A.19. L'unité d'exécution peut être utilisée par 8 *ports* différents. Chaque port est relié à des composants différents de l'ALU qui peut exécuter jusqu'à 4 instructions par cycles (ou 2 opérations flottantes).

A.2.4.3 Processeur multicoeur

La vitesse de calcul des processeurs est liée à sa fréquence. Cette dernière a largement contribué à l'évolution de leur performance. Cependant, certaines limites physiques empêchent l'augmentation infinie des fréquences (discuté dans la partie A.2.3.1). Il a donc fallu trouver d'autres moyens d'améliorer la performance des processeurs, sans pouvoir accélérer leur fréquence. L'apparition des processeurs multicoeurs est une réponse à ce défi. Pour comprendre leur intérêt, l'analogie suivante peut être utilisée [Tan03] : la construction d'un processeur avec une fréquence de 1000 GHz est probablement impossible. Par contre, l'utilisation de 1000 processeurs avec une fréquence de 1 GHz est possible et permet d'obtenir la même performance. Ce gain de performance peut alors être utilisé pour réduire la fréquence des processeurs. En réduisant la fréquence de 30%, l'énergie nécessaire est réduite de 35% [Mat14]. En utilisant deux coeurs à 70% de la fréquence initiale permet cependant d'obtenir un gain de 140% de la puissance de calcul totale. C'est ce constat qui motive l'utilisation du parallélisme dans

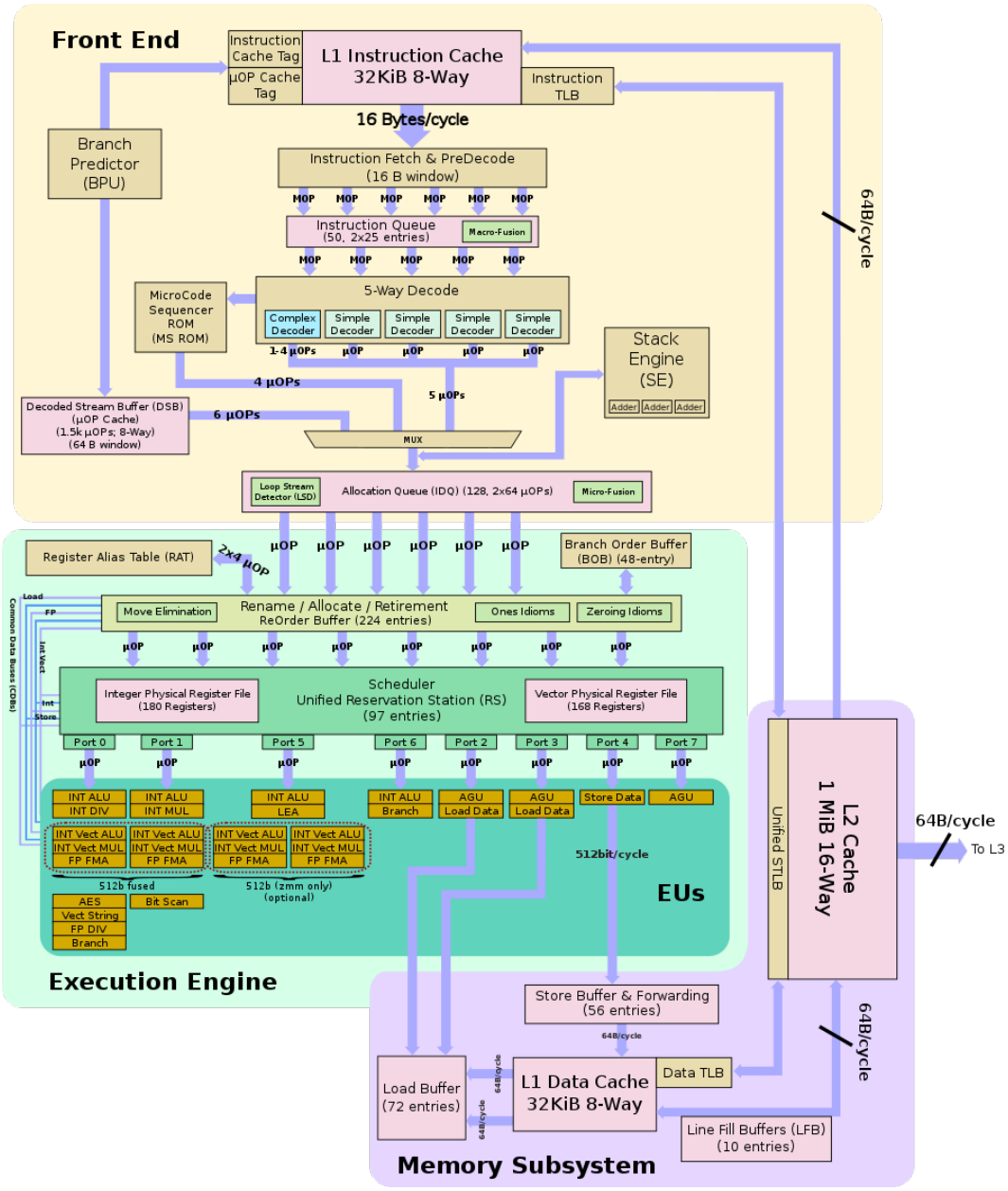


FIGURE A.19 – Microarchitecture des processeurs Skylake.

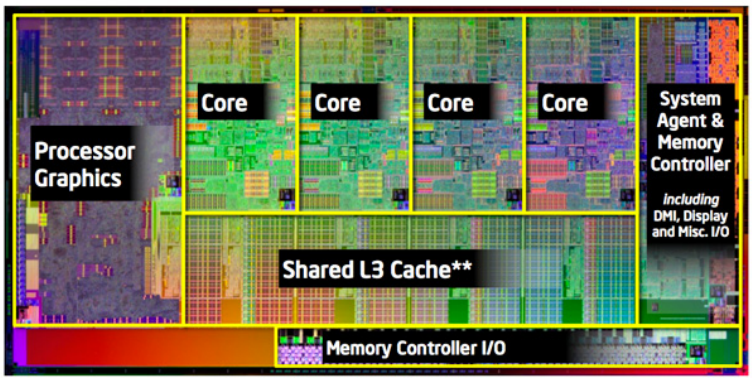


FIGURE A.20 – Exemple de processeur multicoeur (Intel Core i7-2600K) partageant le troisième niveau de cache [8].

toute l'architecture d'un supercalculateur (voir section 2.1.3.3). Ainsi, avant l'apparition des processeurs multicoeurs, l'utilisation du multitraitement symétrique (SMP) utilisant plusieurs processeurs en parallèle était le principal moyen d'accéder au parallélisme. Cependant, les serveurs devenant toujours plus gros, et avec le désir d'avoir des processeurs plus puissant pour les ordinateurs personnels et les téléphones, les processeurs multicoeurs ont été inventés.

Multicoeur. Le terme de processeur multicoeur est employé pour désigner tout processeur possédant entre deux et quelques dizaines de coeurs (on parle de processeurs *manycore* au-delà). Les différents coeurs sont disposés sur la même puce d'où l'autre appellation utilisée pour désigner ces processeurs de Chip Multiprocessor (CMP).

Les différents coeurs sont généralement identiques en tout point (processeur homogène) bien qu'ils puissent être différents (processeur hétérogène). Les processeurs homogènes sont plus faciles à utiliser, car tous les coeurs peuvent répondre au même besoin et leur design est plus simple. Les processeurs hétérogènes sont cependant plus performants pour certaines applications. Généralement, chaque coeur du processeur n'est pas relié directement à la mémoire. Un niveau de cache est généralement interposé entre le coeur et la mémoire. La hiérarchie mémoire est présentée dans la section A.3.

L'avantage principal de dupliquer un coeur plutôt que de doubler la fréquence d'un seul coeur est la consommation électrique et donc la puissance dégagée par effet Joule. En effet, la puissance dissipée est quadruplée quand la fréquence est doublée alors qu'elle ne fait que doubler lorsque le nombre de coeurs est doublé. On obtient ainsi un processeur utilisant moins d'énergie et nécessitant moins de refroidissement pour une même performance qu'un processeur plus rapide. Comme pour le pipeline (voir section A.2.4.1), le gain de performance apporté par l'ajout de coeur s'appuie sur l'amélioration du niveau de parallélisme d'instructions (Instruction Level Parallélisme (ILP)).

La difficulté d'utilisation de processeurs multicoeurs vient des programmes qui ne sont pas

8. source : https://www.anandtech.com/show/4083/the-sandy-bridge-review-intel-core-i7-2600k-i5-2500k-c

capables par nature d'utiliser ce niveau de parallélisme. Ils doivent donc être programmés pour pouvoir en profiter. Cette tâche, quoique difficile à ses débuts, est aujourd'hui facilitée par l'utilisation de librairies prévues telles que *Pthread* ou *OpenMP*.

Les coeurs partagent des niveaux communs de cache (généralement le dernier), la bonne programmation des applications et l'implémentation d'une microarchitecture efficace sont alors des facteurs déterminants de la performance obtenue :

— Pour maximiser l'utilisation des caches, il est primordial de prévoir leurs partages entre les différents coeurs pour minimiser les conflits. Des méthodes de placements plus ou moins efficaces peuvent alors être utilisées [MTB11] : laisser le système d'exploitation s'en occuper peut donner des performances très variables entre deux exécutions identiques, alors que le placement manuel permet d'obtenir les meilleurs résultats.

— Le choix du réseau utilisé par les coeurs est alors important pour la performance des codes et repose principalement sur quatre paramètres [PKV09] : la topologie, les algorithmes de routages, le protocole de contrôle de flux et le routage de la microarchitecture. La topologie indique comment les coeurs sont connectés et quels sont les chemins empruntables par un message pour rejoindre sa destination. Ce choix est réalisé par l'algorithme de routage. Le contrôle de flux s'occupe de l'envoi des messages (ordre et date d'envoi) qui est ensuite réalisé par la microarchitecture. Les choix réalisés pour ces quatre paramètres ont un impact sur la performance du processeur (latence, bande passante) et sur son prix.

Le nombre de coeurs par processeur a beaucoup évolué durant les quinze dernières années. Si les premiers processeurs multicoeurs n'en possédaient que deux, il n'est pas rare que les supercalculateurs utilisent des processeurs avec plus de vingt coeurs. Intel à même annoncé en 2019 un nouveau processeur doté de 56 coeurs [9]. Cependant, l'ajout de coeurs supplémentaires n'est pas forcément bénéfique pour les applications à cause de la vitesse des mémoires qui peinent à évoluer au même rythme. Ce constat est discuté dans la section A.3.1.

A.3 Hiérarchie mémoire

Dans la partie précédente sont présentées les principales améliorations dont la microarchitecture a pu bénéficier. Cette partie s'intéressait principalement à l'exécution des instructions. Pour être exécutées, ces instructions ont besoin de données stockées en mémoire. La performance totale du processeur dépend donc à la fois de la vitesse d'exécution des instructions, mais aussi de la capacité de la microarchitecture à transférer les données nécessaires au processeur. Cette section s'intéresse donc plus précisément à la partie mémoire :

— La section A.3.1 présente la différence d'évolution entre les processeurs et la mémoire.

— Cette introduction permet de motiver la nécessité de construire une hiérarchie mémoire présentée dans la section A.3.2.

9. https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-html

— Enfin, la section A.3.3 présente plus précisément les mémoires caches.

A.3.1 Mémoire principale

Pour leur exécution, les processeurs doivent accéder aux instructions et aux données stockées en mémoire. Celles-ci sont principalement construites à partir de mémoire RAM (voir section A.1.5). Bien que celles-ci aient profité de la miniaturisation des transistors, les composants (voir section A.1.6.1), l'industrie a constaté depuis plusieurs années un écart de performance entre le système mémoire et la puissance de calcul du processeur.

A.3.1.1 Performance de la mémoire et des processeurs

Grâce aux nombreuses améliorations apportées aux processeurs (voir section A.2), l'écart entre leur performance et celle des mémoires s'est creusé au fil des années (voir figure A.21). Ce graphique montre que l'écart de performance entre les deux s'agrandit exponentiellement au fil des ans (utilisation d'une échelle logarithmique en ordonnée) :

— La performance calculatoire des processeurs (FLOP exécutable par cycle) a **augmenté de 50%** en moyenne par an.
— La bande passante entre le processeur et la mémoire a augmenté de 23% par an
— La latence des requêtes mémoires a **diminué de 4%** par an
— La bande passante sur le réseau a **augmenté de 20%** par an

Ainsi l'augmentation du nombre de coeurs, dont la performance augmente (fréquence, pipeline, opérations vectorielles), se partageant le bus mémoire dont les performances évoluent lentement, implique une baisse de bande passante disponible par coeur (voir figure A.22).

A.3.1.2 Le mur de la mémoire

L'écart croissant entre les performances de la mémoire et des processeurs est souvent appelé *mur de la mémoire* (*memory wall*). Cette expression été déjà utilisée en 1995 [WM95] dans le papier *Hitting the Memory Wall : Implications of the Obvious*. Plusieurs raisons peuvent expliquer cet écart de performance.

Une première raison pouvant expliquer l'apparition de cet écart de performance est l'économie du marché des mémoires. Le prix par gigaoctet diminue avec la densité des mémoires. Les fabricants d'ordinateurs et de téléphones souhaitent avoir des mémoires de plus grande capacité dans des espaces restreints avec des contraintes énergétiques. La SRAM, moins dense et plus consommatrice est ainsi passée en second plan. Le gain de transistors assuré par la loi de Moore a été utilisé pour construire des mémoires plus denses et donc de plus grande capacité. Cependant, l'industrie des microprocesseurs a profité de ces transistors pour construire des processeurs plus performants tandis que l'industrie des mémoires en a profité pour créer des mémoires de plus grande capacité.

La seconde raison est technologique. Les différentes évolutions de ces deux technologies mémoire sont résumées dans le tableau A.5. La mémoire DRAM a vu ses performances évo-

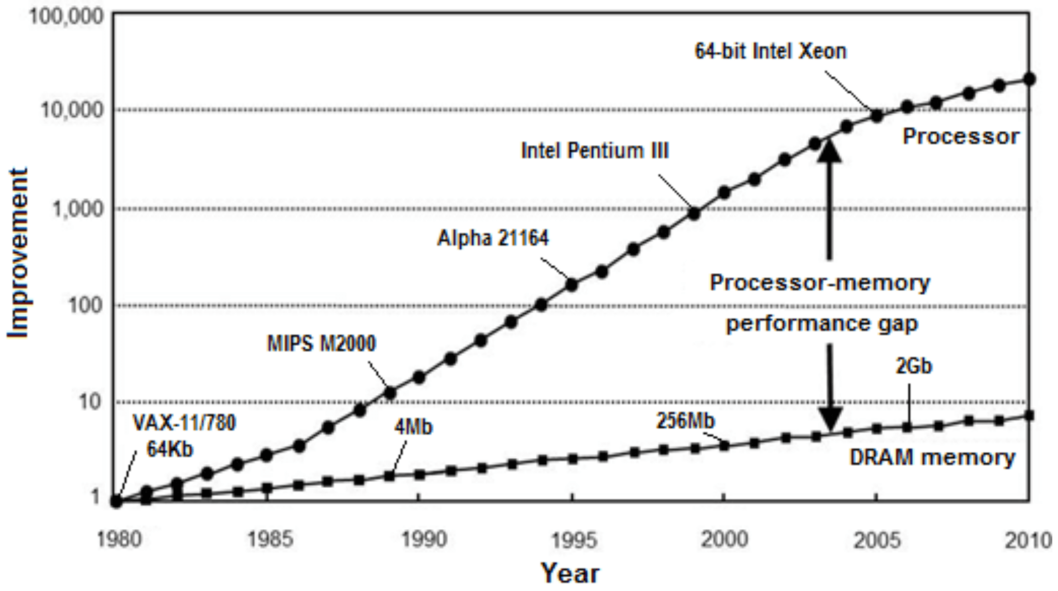


FIGURE A.21 – Progression de la performance des processeurs et des mémoires. La performance de plusieurs générations de processeurs a été mesurée à l'aide du benchmark SPECint [ECT17]. La performance des mémoires est la mesure des latences d'accès mémoire (CAS et RAS) des mémoires DRAM.

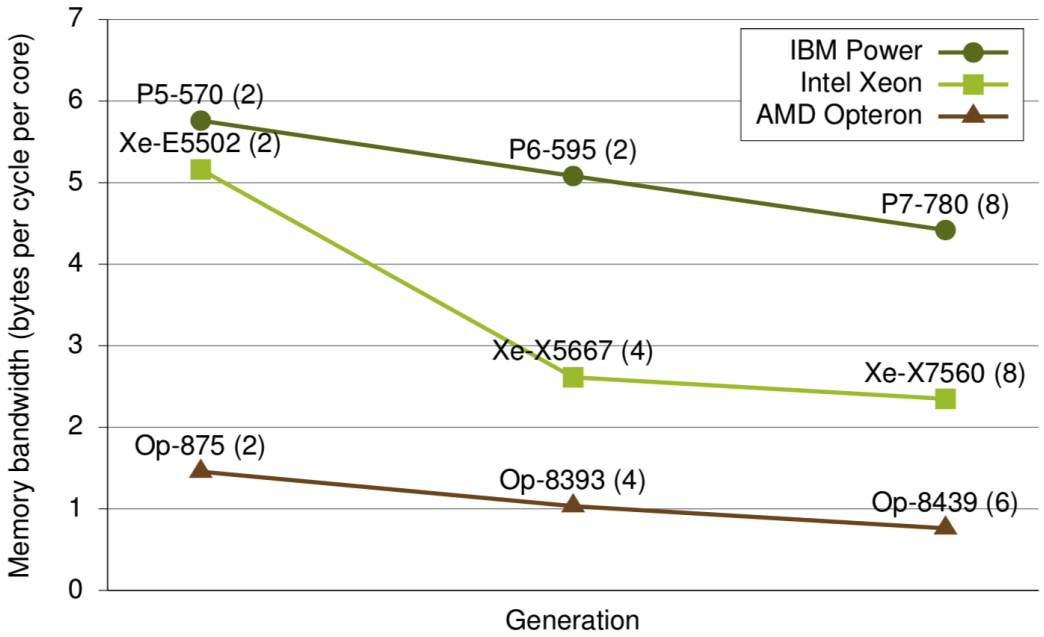


FIGURE A.22 – Évolution de la bande passante bande passante par cycle d'horloge par coeur. L'évolution du nombre de coeurs et de leur fréquence n'est pas compensée par l'évolution de la performance du système mémoire.

DDR Standard	Débit (MT/s)	Densité (MiB)	Bande passante (GB/s)	Latence (ns)	Tension (V)
SDRAM	100-166	1	0.8-1.3	90-75	3.3
DDR	133-200	128	1-3.1	36-30	2.5
DDR2	400-1066	256	3.2-8.3	37.5-14.06	1.8
DDR3	1066-3300	1024	8.3-25.8	19.69-11.82	1.5
DDR4	1600-4800	4096	12.5-37.5	19.38-9.38	1.2

Tableau A.5 – Les évolutions des technologies mémoire DDR ont permis d'améliorer les vitesses de transferts d'un facteur dix en divisant par trois leur consommation électrique [10]. Pour chaque standard DDR, les débits et la latence sont donnés pour deux fréquences différentes. La latence mesure le temps nécessaire pour transférer 8 mots (soit une ligne de cache). Cette mesure est plus précise, car les versions les plus récentes de DDR lisent plusieurs mots à la fois dans une mémoire tampon, et profite donc de la localité spatiale.

luer d'un facteur 10 depuis les premières versions de *SDRAM* dans les années 90. Le passage de la SDRAM à la *Double Data Rate* (DRAM) permet d'envoyer deux données par signal d'horloge. Les principales évolutions des performances de ces technologies sont résumées dans le tableau A.5. Aussi, l'architecture Von Neumann (voir section A.2.1.2) implémente une microarchitecture qui sépare la partie exécution de la partie mémoire. Le processeur a besoin des instructions et des données stockées dans la mémoire ce qui implique de nombreuses communications entre ces deux parties. Le bus mémoire devient un goulot d'étranglement (bottleneck) pour les performances. Ce goulot a ainsi été nommé d'après l'architecte : le *bottleneck Von Neumann*.

A.3.1.3 Latence mémoire

La latence mémoire représente le temps écoulé entre l'initiation d'un transfert par le contrôleur mémoire du processeur et l'arrivée de la donnée sur le processeur. Afin de réduire ce temps, les architectures ont reçu plusieurs améliorations matérielles et logicielles afin de *tolérer* ou *réduire* les latences mémoires [ECT17] :

— **Tolérer** les latences [BGL00] consiste à permettre au processeur de continuer d'exécuter des instructions durant l'attente du transfert d'une donnée. Les processeurs ont reçu plusieurs améliorations matérielles permettant de tolérer l'absence d'une donnée : l'exécution dans le désordre (section A.2.3.5) ou la prédiction de branchement (section A.2.3.5).

— **Réduire** la latence mémoire peut être réalisé de manière logicielle ou matérielle grâce aux techniques de prélecture mémoire (*memory prefteching*) [BC91 ; MG91]. Ces tech-

10. source : https://www.transcend-info.com/Support/FAQ-296, https://promotions.newegg.com/crucial/15-2725/index.html?icid=318028, https://en.wikipedia.org/wiki/CAS_latency https://www.crucial.com/usa/en/memory-performance-speed-latency

Comparing Memory Types

	SRAM	DRAM	ROM	EEPROM	NOR	NAND
Nonvolatile	No	No	Yes	Yes	Yes	Yes
Erasable	Yes	Yes	No	Yes	Yes	Yes
Programmable	Yes	Yes	Factory	Yes	Yes	Yes
Smallest Write	Byte	Byte	N/A	Byte	Byte	Page
Smallest Read	Byte	Page	Byte	Byte	Byte	Page
Read Speed	V Fast	Fast	Fast	Fast	Fast	Slow
Write Speed	V Fast	Fast	N/A	Slow	Slow	Slow
Active Power	High	Med	Med	Med	Med	Med
Sleep Power	V Low	High	Zero	Zero	Zero	Zero
Price/GB	High	Low	V Low	High	Low	V Low
Applications	Small Fast	Main Memory	Stable Code Volume	Serial #, Trim	Code	Data

Objective Analysis
Semiconductor
Market Research

EE Times techonline

FIGURE A.23 – Comparaison des caractéristiques de différents types de mémoires.

niques ont pour objectif de prédire les futurs accès mémoire afin de réduire ou supprimer la latence lors d'un accès mémoire. La prélecture peut être réalisée par le système d'exploitation ou par le compilateur en se basant sur des motifs d'accès pour anticiper les prochains appels mémoires. Elle peut aussi être implémentée manuellement dans l'application en utilisant des instructions spécifiques ou en anticipant les accès.

A.3.2 La hiérarchie mémoire

A.3.2.1 Compromis entre coût, performance et capacité

La réponse naïve au problème du *memory wall* est de construire de grandes mémoires à partir de SRAM, très performantes et qui consomment peu d'énergie. Cependant, des contraintes économiques et techniques sont à prendre en compte et rendent impossible cette solution. La mémoire SRAM est très chère à produire et nécessite l'utilisation de six transistors pour fonctionner, empêchant la construction de modules denses. Dans le tableau A.23, on remarque que trois caractéristiques d'une mémoire sont liées :

1. Plus sa latence d'accès est rapide, plus son prix est élevé,
2. Plus sa capacité mémoire est élevée, plus son prix par bit est réduit.
3. Plus sa capacité est élevée, plus sa latence d'accès est longue.

L'objectif des constructeurs d'ordinateur est de proposer une mémoire de grande capacité, très rapide à un prix le plus faible possible, ce qui n'est malheureusement le cas d'aucune des technologies existantes. Pour atteindre cet objectif, la solution imaginée par les architectes est l'utilisation de différents tailles et types de mémoires constituant une *hiérarchie de mémoires*.

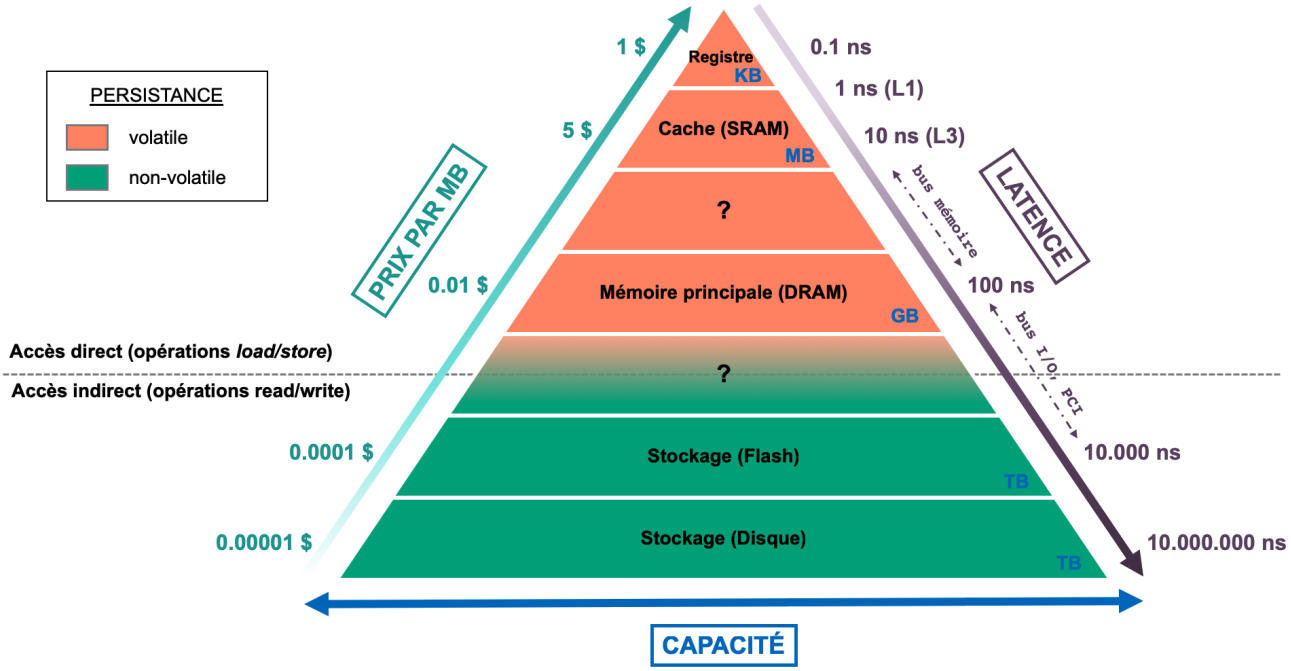


FIGURE A.24 – Hiérarchisation des différents types de mémoire en fonction du coût, de la latence et de la capacité de stockage habituellement utilisés dans les architectures modernes. Les écarts de performance relevés dans la hiérarchie mémoire sont modélisés par un '?'.

A.3.2.2 Description d'une hiérarchie mémoire

La hiérarchie mémoire est la réponse d'un compromis entre le coût, la performance et la capacité de stockage pour la construction de la mémoire d'un processeur. Comme aucune des mémoires ne répond à ces trois facteurs à la fois, la solution est de ne pas se restreindre à une seule technologie et d'utiliser des modules de tailles différentes (voir figure A.24). Le compromis réalisé par la hiérarchie mémoire permet ainsi de : réduire le coût par bit stocké, augmenter la capacité de la mémoire et améliorer le temps d'accès aux données.

La solution est de placer au plus proche du processeur des mémoires très rapides pouvant répondre instantanément aux accès mémoire. Plus on s'éloigne des unités de calcul, plus la latence d'accès aux modules mémoires augmente, mais plus leur prix diminue rendant possible l'utilisation de module de plus grande capacité. Lorsque le processeur souhaite accéder à une donnée, il vérifie qu'elle se trouve dans son premier niveau de mémoire et, si ce n'est pas le cas, remonte la hiérarchie jusqu'à la trouver. La performance des applications varie fortement si les données nécessaires sont présentes ou non dans les mémoires proches du processeur. L'apport d'un gain de performance de la hiérarchie réside dans le fait que la donnée accédée doit se trouver le plus souvent possible dans les zones mémoires les plus proches du processeur. La figure A.24 présente les principaux modules d'une hiérarchie mémoire d'un processeur moderne :

1. **Les registres** : ils sont situés au plus proche des unités de calculs. Pour permettre un accès rapide, inférieurs à 1 cycle, les registres sont réalisés en SRAM. On compte une

centaine de registres sur les processeurs récents. Leur taille est variable en fonction des instructions exécutables par les unités logiques arithmétiques. Par exemple, un processeur pouvant exécuter des instructions vectorielles AVX-512, possède des registres de 512 bits (registre *ZMM*). Il existe différents types de registres, certains sont utilisés pour stocker des données et des résultats intermédiaires, tandis que d'autres ont une signification précise. Le registre de pointeurs de piles stocke l'adresse de la première adresse mémoire responsable de l'instruction actuellement exécutée et permet de réaliser des appels et des retours de fonctions. Les registres des drapeaux (Flag Register) stockent des informations nécessaires à l'exécution d'instructions. Par exemple, lorsqu'une retenue est générée par un calcul, ou qu'un branchement conditionnel a été évalué à vrai. Les processeurs récents dupliquent certains registres pour pouvoir utiliser des techniques de renommage [MPV93] et d'exécution spéculative [CA04].

2. **Les caches** ou *antémémoire*s sont des modules de mémoire généralement en SRAM à très faible latence d'accès (quelques cycles). Ces mémoires étant très chères, elles ne peuvent pas être construites en grande quantité. La performance d'une application dépendra fortement de la présence ou non des données nécessaires dans ces mémoires. Ce principe est connu sous le nom de concept de *localité*, présenté dans la section A.3.2.3. Les caches sont présentés plus précisément dans la section A.3.3.

3. **La mémoire**, ou mémoire principale, stocke les instructions du programme ainsi que les données nécessaires à leur exécution. Pour des grands jeux de données dépassant la taille de la mémoire, cette dernière agit comme un cache entre le processeur et les modules de stockage.

4. **Le stockage** est le premier étage de la hiérarchie à utiliser une mémoire non volatile. Ces modules peuvent contenir des téraoctets de données, mais dont les temps d'accès sont extrêmement longs par rapport aux mémoires mentionnées ci-dessus. Leur principal avantage est leur prix.

5. **La mémoire auxiliaire** est aussi non volatile avec un coût bien plus faible que pour les mémoires de stockage. Sa capacité à stocker les données pendant plusieurs années est utilisée pour stocker et archiver des données de nécessitant pas d'accès immédiat. Les supports sont magnétiques (disques et bandes) ou optiques.

Communication entre les différents niveaux Pour communiquer entre les différents niveaux de la hiérarchie, les données sont transmises par blocs de données de tailles différentes (voir figure A.25). L'avantage de transférer les données par blocs et non une par une est d'améliorer la performance des codes en tirant parti du principe de localité spatiale (voir section A.3.2.3). Pour accéder à un mot, le processeur a besoin que celle-ci se trouve dans le niveau de cache L1. Lorsqu'elle s'y trouve, le processeur peut charger un mot directement dans ces registres pour y effectuer les opérations nécessaires. C'est la granularité de transfert la plus petite dans une microarchitecture. Entre les différents niveaux de caches et entre le cache et la mémoire les données sont transférées par blocs appelés *lignes de cache* ou ligne de cache. La ligne de cache

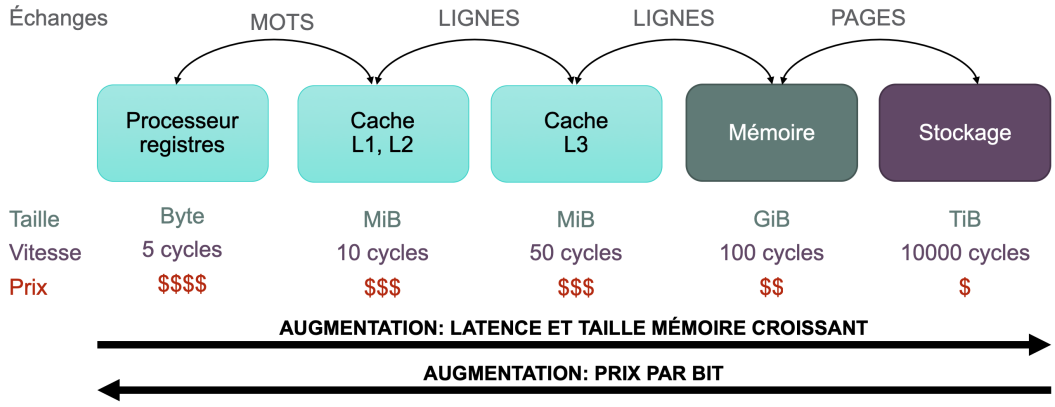


FIGURE A.25 – Hiérarchie mémoire

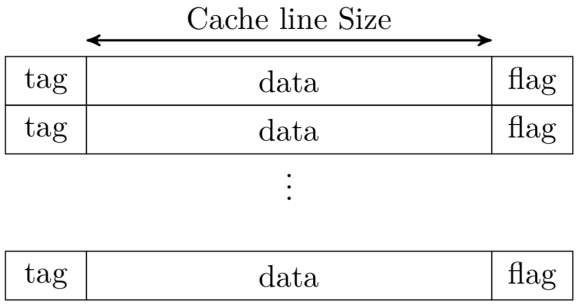


FIGURE A.26 – Représentation d'une ligne de cache.

contient une copie des données de la mémoire, un tag contenant des informations sur l'adresse mémoire du bloc de données et un *flag* contenant des informations sur la validité de la ligne (voir figure A.26). La taille d'une ligne de cache peut varier d'une architecture à l'autre, mais il est courant d'utiliser des tailles de 32, 64 ou 128 octets. Une ligne de cache d'un processeur Intel récent mesure 64 octets. Elle contient ainsi 8 éléments en double précision. Enfin, entre le stockage et la mémoire, les blocs de données transférés sont de la taille d'une page (voir section A.4.1) qui mesure généralement entre 4 KiB et 2 MiB.

A.3.2.3 Localité

Mal utilisée, la hiérarchie mémoire n'apporte aucun gain de performance. Son efficacité réside dans la présence des données utilisées dans les niveaux de caches les plus proches de la mémoire. Afin d'améliorer la performance des applications, il est nécessaire de profiter de la localité des données. Le principe de localité exprime la faculté d'un programme à utiliser des données présente dans les premiers niveaux de cache. Elle peut être de deux types :

— **La localité spatiale** s'appuie sur l'utilisation de données dont les adresses sont consécutives en mémoire (accès encadré en vert sur la figure A.27). En effet, la taille d'un bloc de données transféré (une ligne de cache), mesure plusieurs octets et contient plusieurs

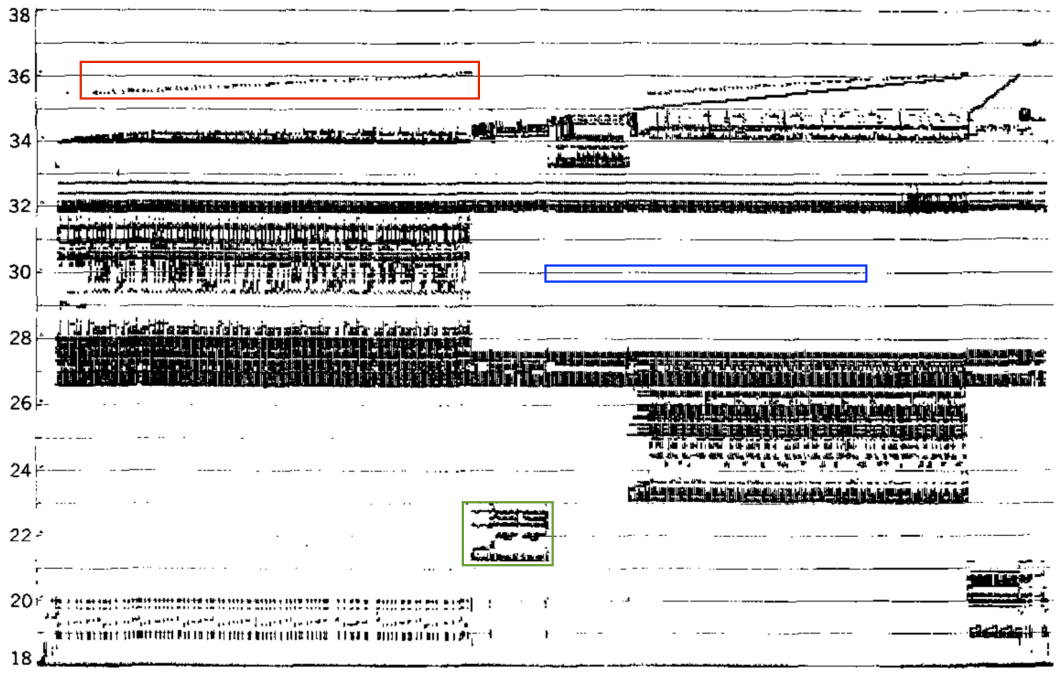


FIGURE A.27 – Évolution du numéro des pages accédé (axe des ordonnées) en fonction du temps (axe des abscisses) tiré de [HG71]. Les accès encadrés en bleu ont une localité temporelle. Les accès encadrés en vert ont une localité spatiale. Les accès encadrés en rouge ne profitent d'aucune localité.

éléments. Un programme utilisant des données consécutives en mémoire, aura la chance de trouver la donnée suivante déjà chargée dans le cache. Par exemple, les instructions d'un programme se suivent en mémoire et sont accédées consécutivement.

— **La localité temporelle** exprime la réutilisation d'une donnée dans un futur proche, augmentant la probabilité qu'elle soit toujours dans le cache. Ces accès sont encadrés en bleu sur la figure A.27).

Le programmeur doit être conscient de ces deux opportunités lorsqu'il écrit son programme. Pour profiter de la localité spatiale, il doit organiser ses données de telle sorte que l'algorithme les parcourt de façon continue. Si la taille de la mémoire le permet, il peut être avantageux de transformer la structure de données, de façon temporaire, pour placer les données utilisées consécutivement en mémoire. Pour profiter de la localité temporelle, le programmeur doit veiller à que le maximum d'opérations soient exécutées sur une ligne de cache lorsqu'elles se trouvent dans la mémoire cache. Il peut alors être utile de réorganiser une boucle de traitement. Cependant, des applications ne peuvent pas, par nature, profiter des localités temporelle et spatiale comme en cryptographie ou analyse de signaux. Ces applications accèdent à des jeux de données immenses de façon aléatoire ne pouvant pas tenir dans les caches, rendant ces derniers inutiles.

A.3.2.4 Saturation du bus mémoire

Les performances des applications étant principalement limitées par le bus mémoire, il est important que ce dernier soit utilisé à son maximum. On parle alors de *saturation* du bus mémoire. Pour comprendre le lien entre la saturation du bus et la latence mémoire nous utilisons la loi de Little .

Loi de Little [Lit61]. Cette loi énoncée en 1961 est utilisée dans la théorie des files d'attente. Elle sert à caractériser le nombre de personnes N dans un système stable (comme une file d'attente), en fonction du temps de traitement L et du débit moyen de client T. Le nombre de personnes dans la file peut être calculé par la loi de Little (voir équation A.2). Ainsi, pour réduire le temps d'attente d'un client dans une file, il faut soit réduire le nombre de personnes présentes simultanément dans la queue, ou augmenter la cadence de traitement.

$$N = L \times T \tag{A.2}$$

Bien qu'en apparence éloignée du domaine des ordinateurs, cette loi est fondamentale pour comprendre la performance du bus mémoire. Le bus mémoire peut être vu comme une file d'attente dont :

— les clients sont des requêtes mémoires en attente ;
— le temps de traitement est la latence mémoire ;
— le débit représente la bande passante mémoire.

La latence et le débit mémoire étant des caractéristiques de l'architecture, le but du programmeur est d'utiliser la performance maximale du bus mémoire. Il doit pour cela générer suffisamment de transactions mémoires permettant sa saturation.

Prenons l'exemple d'un bus mémoire ayant une bande passante de 128 GB/s transférant des lignes de cache de 64 bytes et une latence mémoire de 100 ns. Son débit est alors de 2 lignes de cache par nanoseconde. Avec la loi de Little, il est possible de calculer le nombre de requêtes mémoires concurrentes nécessaires pour saturer la bande passante mémoire. Dans notre exemple, il est égal à $100 \times 2 = 200$ transactions (voir figure A.28). Si le code, ou l'architecture n'est pas capable de gérer, au minimum, 200 requêtes mémoires *en vol* (*outstanding load*), le bus mémoire ne pourra jamais être saturé. Un autre impact est que si un processeur est capable de les générer, lui ajouter des coeurs, ne permettra pas d'obtenir de meilleures performances du bus. Pour certains codes limités par la bande passante, il peut être intéressant d'éteindre certains coeurs pour limiter la consommation du processeur sans affecter la performance du code.

Cette loi à un impact direct sur les architectures modernes utilisant des mémoires HBM. Ces mémoires ont des latences d'accès un peu plus faible dû à leur proximité du processeur. Les débits mémoires sont eux aussi bien meilleurs. La loi de Little, implique que le nombre d'accès mémoire concurrents nécessaires pour saturer ces bus va fortement augmenter. Malheureusement, la microarchitecture actuelle des processeurs n'est pas forcément adaptée à de telles évolutions. Ainsi, nous avons rencontré des architectures incapables de saturer la bande

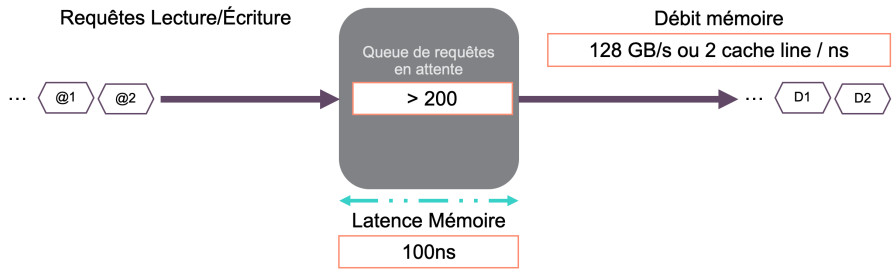


FIGURE A.28 – La loi de Little permet de calculer le nombre de requêtes mémoires nécessaire pour saturer le bus mémoire. Pour un bus mémoire de 128 GB/s et une latence mémoire de 100 ns, il faut qu'il y ait plus de 200 requêtes mémoires en attente pour que le bus soit saturé.

passante mémoire disponible.

A.3.3 Caches

C'est en 1965 que les premières mémoires caches sont présentées sous le nom de *slave memory* [Wil65]. Leur temps d'accès est 4 à 20 fois plus rapide que celui de la mémoire principale. Cependant, leur taille est très réduite (quelques MiB) comparée à celle de la mémoire principale (plusieurs GiB). Les caches utilisent généralement de la mémoire SRAM.

Ce niveau de mémoire a été implémenté pour réduire, du point de vue du processeur, l'écart de performance entre ses unités de calculs et celles de la mémoire centrale. Leur apport de performance vient de la capacité des programmes à réutiliser des données déjà présentes dans le cache, évitant un accès à la mémoire centrale beaucoup plus long. Lorsque le processeur doit accéder à une donnée, il commence par la chercher dans le premier niveau de cache, si elle s'y trouve, son temps d'accès est très rapide (événement *cache hit*). Si ce n'est pas le cas (événement *cache miss*), il réalise alors une copie de la zone mémoire la contenant dans le cache. La zone mémoire copiée est appelée *ligne de cache*. Si par la suite, cette donnée ou une donnée appartenant à la même ligne de cache devait être à nouveau accédée, leur temps d'accès serait alors drastiquement réduit. Ce mécanisme est transparent pour l'utilisateur, bien que pour des questions de performances il doive être conscient de son existence (voir section A.3.2.3).

La taille de chaque niveau de cache varie pour les raisons expliquées en introduction de cette partie. À cela vient s'ajouter la notion de performance qui est liée à leur taille. Un cache de grande capacité aura plus de chance de contenir la donnée dont le processeur a besoin, améliorant ainsi la performance moyenne du programme.

Pour allier les avantages et contourner les inconvénients, les processeurs utilisent non pas un, mais plusieurs niveaux de caches de tailles différentes. Le premier niveau de cache est généralement séparé en deux zones mémoire : l'une contenant les instructions et l'autre les données. C'est le seul niveau de la hiérarchie qui stocke différemment les données et les instructions. Sur les processeurs récents, le premier et le deuxième niveau de cache est privé à chaque coeur. Un troisième et parfois un quatrième niveau de cache sont partagés entre les différents coeurs du

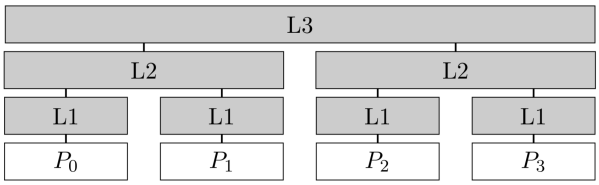


FIGURE A.29 – Organisation d'une hiérarchie de caches à trois niveaux sur un processeur à 4 coeurs (source [Put14]).

processeur. La figure A.29 représente une telle architecture pour un processeur à 4 coeurs. Le partage d'un ou plusieurs niveaux de caches entre différents coeurs à certains bénéfices en programmation parallèle. La communication entre les coeurs est plus rapide, ainsi que la migration d'un thread entre deux coeurs partageant un même niveau de cache. Cependant, cela introduit de la complexité pour la cohérence des caches (voir section A.3.3.5).

A.3.3.1 Propriété d'inclusion

Lorsqu'une donnée est chargée depuis la mémoire, le processeur doit la stocker dans le cache. La manière de stocker l'information varie en fonction de la propriété d'inclusion du processeur :

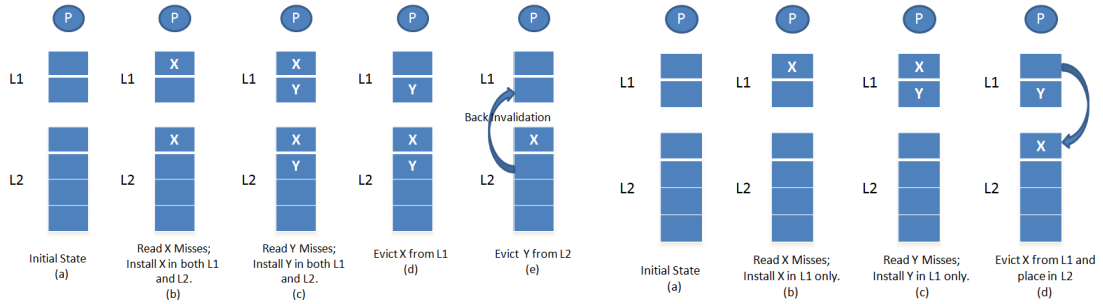
— Un cache est dit inclusif, si une donnée peut se trouver dans plusieurs niveaux de cache au même moment (voir figure A.30a). Cette politique d'inclusion a un désavantage lorsqu'elle est utilisée sur des systèmes multicoeurs. En effet, lorsqu'une donnée doit être retirée d'un niveau de cache, le processeur doit aussi l'enlever des niveaux de caches inférieurs alors que celle-ci pouvait être actuellement utilisée par un autre coeur.

— Un cache est dit exclusif, si à un instant donné, une donnée ne peut pas être stockée à plusieurs endroits dans la hiérarchie de cache (voir figure A.30b). L'avantage des caches exclusif est leur capacité de stocker plus de données, car une ligne de cache ne se trouve jamais à deux endroits à la fois de la hiérarchie de cache. Cependant, lors d'un *hit* dans le cache L2, le processeur doit échanger la ligne entre les deux niveaux de cache L1 et L2, plus long qu'une simple copie.

Les processeurs récents peuvent implémententer les deux politiques d'inclusion. Le processeur Intel Sandy Bridge a un cache L3 inclusif tandis que les caches L1 et L2 sont non-inclusif.

A.3.3.2 Politique de placement : associativité

La performance d'un cache ne vient pas seulement de la technologie utilisée pour sa construction. En effet, lorsqu'une donnée est accédée, le cache doit vérifier si la donnée est présente ou non dans le niveau de cache demandé. Il faut que l'algorithme de comparaison permettant de la trouver soit le plus rapide possible. Pour cela, les architectes utilisent généralement une fonction de *hash* permettant d'attribuer un emplacement dans le cache en fonction de l'adresse mémoire de la ligne de cache. Si la ligne de cache ne se trouve pas à l'emplacement (ou les emplacements) calculés, c'est qu'elle n'est pas présente dans ce niveau de cache. Les trois politiques de placement les plus utilisées sont :



(a) Inclusive (b) Exclusive

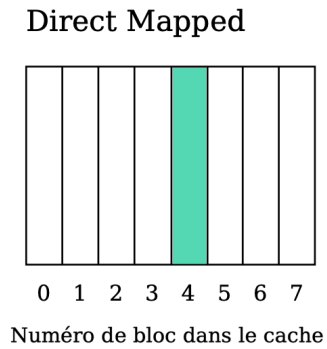
FIGURE A.30 – Exemple de deux propriétés d'inclusion de la hiérarchie de cache (source [Wik19b]).

— le mappage *direct* - figure A.31
— le mappage *fully associative* - figure A.32
— le mappage *set assiociative* - figure A.33

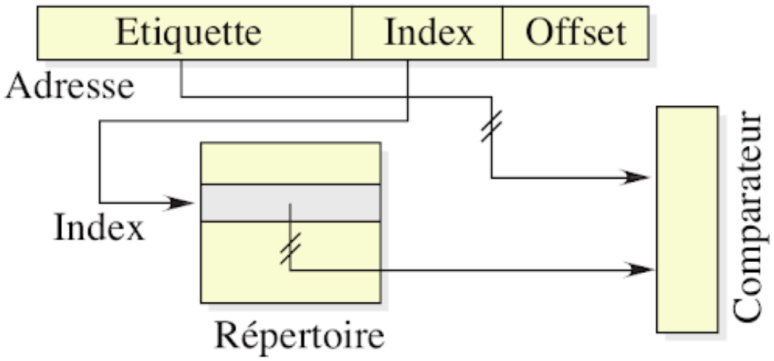
Cache à correspondance directe (*direct-mapped cache*) Cette politique utilise une fonction simple pour déterminer l'emplacement du cache à utiliser. Une partie des bits de l'adresse de la ligne de cache est utilisée pour déterminer la ligne à utiliser (*index*) par exemple à l'aide d'une opération modulo. L'autre partie est utilisée pour déterminer le décalage dans cette ligne (*offset*) (voir figure A.34). Cette méthode est très rapide, mais peut avoir des performances catastrophiques. Un algorithme faisant des sauts en mémoire d'une certaine taille pourrait n'utiliser qu'une seule ligne du cache, le rendant totalement inefficace. Comparé aux caches associatifs, le mappage direct est plus simple à implémenter, car un seul comparateur est nécessaire pour déterminer la ligne de cache à utiliser (voir figure A.34b).

Cache pleinement associatif (*fully associative cache*) Cette politique remédie à l'inconvénient relevé concernant la correspondance directe, en permettant à une ligne de cache d'être stockée à n'importe quel emplacement (figure A.32). Ainsi, la totalité du cache peut être utilisée. Cependant, cette technique a le désavantage d'être très lente. En effet, une ligne de cache pouvant se trouver à n'importe quel index du cache, il faut tous les comparer pour vérifier sa présence ou non. Pour faire cette comparaison en parallèle, il faudrait implémenter autant de comparateurs que de ligne dans le cache, complexifiant grandement le cache.

Cache N-associatif (*N-way set associative cache*) Cette politique permet de réduire le nombre de comparateurs qu'un cache associatif devrait posséder pour être rapide. Ceci est réalisé en regroupant les lignes du cache potentiellement adressable pour une ligne de cache en groupe (*set*). L'exemple de la figure A.33a utilise un cache à 4 set regroupant 2 lignes de cache, appelé *2-way associative*. Il ne faut plus que 4 comparateurs pour déterminer si une ligne appartient à un des *set*. Les mappages par association sont plus lents que le mappage direct,

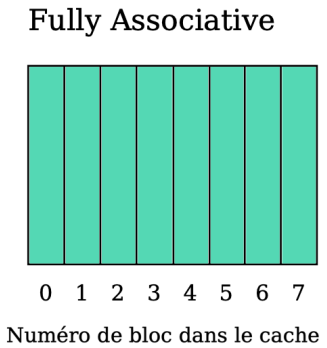


(a) Un seul emplacement par ligne de cache [Meu17]).

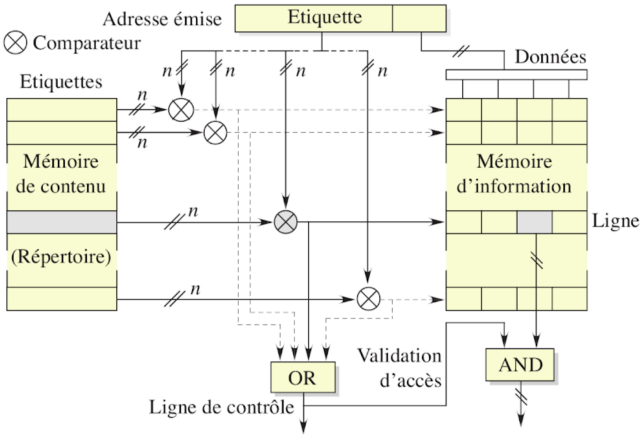


(b) Le cache pleinement associatif nécessite d'avoir un comparateur pour chacune des lignes du cache (extrait de l'ouvrage [BDI13]).

FIGURE A.31 – Cache à correspondance directe

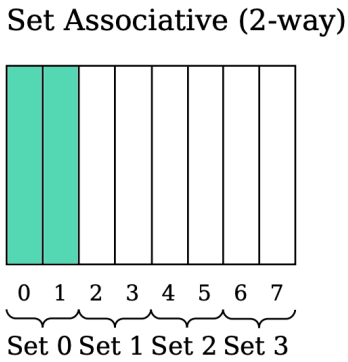


(a) Une ligne de cache peut être à n'importe quel emplacement [Meu17]).

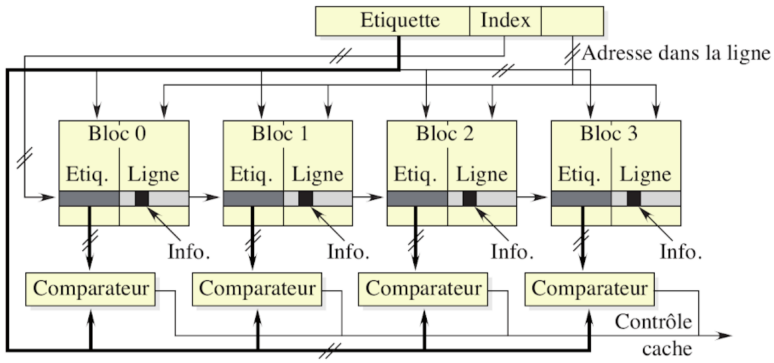


(b) Le cache pleinement associatif nécessite d'avoir un comparateur pour chacune des lignes du cache (extrait de l'ouvrage [BDI13]).

FIGURE A.32 – Cache pleinement associatif

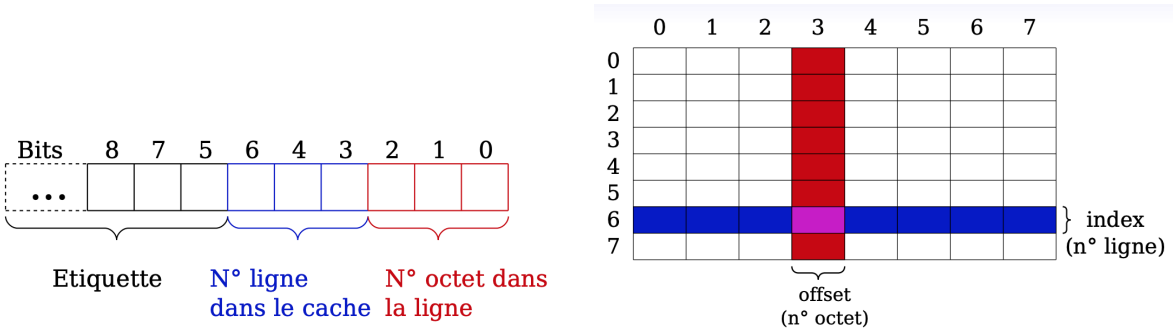


(a) Une ligne de cache peut être à un nombre restreint d'emplacements [Meu17])



(b) Le cache N-associatif associe les deux architectures des caches direct et pleinement associatif. Ce sont plusieurs caches directs montés en parallèle (extrait de l'ouvrage [BDI13]).

FIGURE A.33 – Cache N-associatif



(a) Calcul de l'emplacement (index de la ligne et décalage) pour un mappage direct

(b) Emplacement de la ligne calculée dans le cache ou sera stockée la ligne de cache.

FIGURE A.34 – Exemple du calcul de l'emplacement de la ligne de cache lors d'un mappage direct à partir de l'adresse de la ligne de cache à stocker (source [Meu17]).

car il faut trouver où se trouve la ligne de cache (si présente) dans un sous ensemble de ligne de cache plus ou moins grand. Pour accélérer la recherche de la présence ou non d'une ligne de cache, le traitement peut être réalisé en parallèle dans les différents *sets* par l'utilisation de plusieurs comparateurs (4 dans la figure A.33b).

La propriété principale d'un cache est sa capacité à conserver les données pour de futurs accès. Un cache de 8 MB à 2 sets associatifs peut sauver jusqu'à 44% des *miss* comparé à un cache à correspondance direct [Dre07]. Sur les architectures récentes Intel Skylake, les caches utilisent entre 8 et 16 associativités, pouvant varier entre les différents niveaux.

A.3.3.3 Politique de remplacement

Que ce soit pour le mappage *fully associative* ou *set associative*, une ligne de cache peut être stockée dans plusieurs lignes du cache. Pour déterminer laquelle choisir pour y placer la ligne de cache, différentes stratégies peuvent être utilisées, appelées *politique de remplacement*. L'objectif de ces politiques est de maximiser l'utilisation du cache en prévoyant et en anticipant les futures lignes a être accédées pour ne pas les supprimer du cache. Il existe de nombreuses politiques de remplacement, chacune ayant ses avantages et ses inconvénients [Wik19c] : FIFO, LIFO, LRU, TLRU, MRU, PLRU, RR, SLRU, LFU, LFRU, LFUDA, LIRS, ARC, CAR, MQ. La politique choisie à un réel impacte sur les performances de l'application. Pour la choisir, il faut trouver un compromis entre la performance et la complexité apportée par sa mise en place. Dans le cas du mappage *direct*, la ligne de cache présente à l'emplacement calculé est forcément remplacée et ne nécessite pas d'avoir une politique de remplacement. Il existe deux familles de politiques de remplacement. La première famille regroupe les politiques de remplacements qui tiennent compte de l'utilisation des ligne de cache (LRU, FIFO). Ce sont généralement les politiques les plus efficaces. La deuxième famille est celle des politiques aléatoires (*random*, *round robin*) qui ne tiennent pas compte de l'utilisation des données et choisissent une ligne aléatoirement à remplacer. Ces politiques sont performantes en termes de rapidité d'exécution,

car le choix ne se fait que sur une fonction aléatoire. Cependant, [AMM04] montrent que ces techniques utilisent 22% moins bien le cache, impactant les performances des applications.

Least Recently Used ou LRU remplace la ligne de cache la moins récemment utilisée. Les numéros des lignes utilisées sont stockés dans une pile suivant la date de leur dernière utilisation. La pile est mise à jour lorsqu'une nouvelle donnée est stockée dans le cache en empilant son adresse au sommet de la pile. De même, lors d'un *hit*, la ligne de cache référencé est stockée elle aussi en sommet de pile. Cette méthode à un inconvénient pour certains types d'accès, notamment les parcours de tableau. Imaginons un cache pouvant contenir 4 lignes. Le double parcours d'un tableau mesurant 5 lignes de caches a,b,c,d,e aura les accès mémoire suivants : a,b,c,d,e a,b,c,d,e. Le deuxième accès au tableau ne profitera pas du cache, car chaque ligne de cache est remplacée au fur et mesure du parcours du tableau. Des améliorations ont été apportées pour corriger ce problème, comme l'introduction d'un répertoire image [Sto87] qui garde une trace des groupes de lignes de cache utilisées ensemble pour prévoir les accès similaires et anticiper leur accès.

A.3.3.4 Stratégie de cache : lecture et écriture

Le cache est une zone mémoire qui évolue en fonction des accès mémoires. Son fonctionnement lors d'un accès (en lecture ou écriture) peut varier en fonction de la présence (*hit*) ou non (*miss*) de la donnée et de la stratégie de cache implémentée.

Lecture : lorsque le processeur accède à une donnée, il vérifie qu'elle n'est pas présente dans ses différents niveaux de cache. Si la donnée est présente, son accès est très rapide. Si ce n'est pas le cas, il réalise alors une copie de la zone mémoire la contenant dans le cache (la taille de la zone est une ligne de cache).

Écriture : le comportement du cache lors d'une écriture dépend de la présence ou non de la *ligne de cache* et la politique employée.

Si la ligne de cache n'est pas présente (*miss*) dans le cache, deux solutions sont possibles. La première est de charger la ligne depuis la mémoire et d'y apporter les modifications (politique *Write-Allocate*). La deuxième solution est d'écrire la ligne de cache sans la charger (politique *No-Write-Allocate*). La ligne de cache ne sera chargée que lors d'un miss lors d'une lecture, sauf si la totalité de la ligne a été écrite, les données originales n'ayant plus de valeur utile. Cette option peut être intéressante si un algorithme ne fait qu'écrire dans une structure de donnée sans ne jamais la lire.

Si la ligne de cache est présente (*hit*) dans le cache, deux solutions sont possibles. La première est de mettre à jour la ligne de cache dans le cache et en mémoire pour que le changement soit répercuté sur l'ensemble de la hiérarchie mémoire (politique *Write-Through*). Cette politique peut être pénalisante si le processeur effectue consécutivement la mise à jour d'une donnée (par exemple un compteur, ou un index de boucle). La seconde solution est de différer l'écriture à

plus tard (politique *Write-Back*). L'écriture est effectuée seulement dans le cache et ne sera effective en mémoire seulement lorsque la ligne de cache modifiée sera évincée du cache. La ligne de cache modifiée est alors indiquée grâce à un bit indicatif (*dirty bit*). Comparée à la première méthode, celle-ci utilise moins de bande passante, car les mises à jour en mémoire sont moins fréquentes. Cependant, si plusieurs coeurs utilisent la même donnée, sa valeur pourrait alors être différente entre leurs caches respectifs (donnée périmée). Il faut alors implémenter un protocole de cohérence de cache entre les différents caches et la mémoire.

Les politiques utilisées lors d'un *miss* ou d'un *hit* peuvent être associés. Les combinaisons les plus utilisées sont *Write-Through* + *No-Write-Allocate* et *Write-Back* + *Write-Allocate*.

A.3.3.5 Cohérence de cache

La stratégie employée lors de la modification d'une donnée introduit un challenge majeur des architectures multicoeurs qui est de garantir la cohérence des données entre les différentes zones mémoires. Lors d'un accès mémoire, on souhaite accéder à la valeur la plus récente, qui aurait pu être modifiée par un autre coeur, ou processeur. La gestion de la cohérence d'un processeur à un seul coeur est plus simple, bien qu'elle doive tout de même être implémentée. Les opérations d'entrée-sortie peuvent affecter des données en mémoire qui se trouvent aussi dans les caches.

Le protocole de cohérence de cache est responsable de vérifier qu'une même ligne de cache présente à plusieurs emplacements de la mémoire est identique. Il doit pour cela garantir trois points. Le premier est de partager le changement d'une valeur à tous les coeurs d'un processeur pour que l'ordre des opérations affecté à une valeur soit vu dans le même ordre par tous les coeurs/processeurs. Le deuxième point est d'assurer que le résultat ne dépende que de l'ordre des instructions du programme assembleur et non de l'ordre de leur exécution par les différents coeurs. Enfin, le protocole doit assurer à un coeur qui lit une donnée que sa valeur est bien la dernière qui a été écrite (par un autre coeur ou autre processeur). La notion d'ancienneté peut être définie de plusieurs façons et le protocole doit la définir précisément pour assurer la validité des résultats [BDI13]. En effet, l'ordre peut faire référence à l'ordre des instructions dans le programme source. L'ordre peut aussi faire référence à celui de la fin des exécutions des résultats (avant que la donnée ne soit effectivement écrite). Enfin, ce peut être l'ordre des écritures mémoires. Comme la durée de propagation des écritures n'est pas constante dans le système, des erreurs peuvent apparaître si un protocole venait à utiliser ce dernier.

Comme le résume [BDI13], les deux propriétés principales d'un protocole de cohérence sont sa simplicité de mise en oeuvre et sa performance. Pour assurer la cohérence, deux familles de protocoles existent, suivant si la gestion de cohérence est répartie sur les différents caches (*locale*), ou si elle est centralisée (*globale*).

Protocoles locaux - cohérence répartie Les protocoles *locaux* utilisent des outils de scrutation (*snooping*) et de signalisation (*broadcasting*). Implémentés directement dans les caches, ils ne nécessitent pas la modification ni de la mémoire ni du processeur. Lorsqu'une ligne de

cache est modifiée dans un cache, il obtient la copie exclusive de celle-ci en invalidant ses copies dans d'autres caches (*Write-Invalidate*). Une seconde option vise à simplement signaler la modification de cette ligne aux autres caches pour qu'ils mettent à jour leur structure de donnée (*Write-Update*). Différents protocoles de cohérence ont été implémentés et ont évolué. Les plus connues sont les protocoles MESI (ou *Illinois*) [PP84] et MOESI. Mais il en existe beaucoup d'autres : MSI, MOSI, MERSI, MESIF. MESI et MOESI sont notamment très utilisés dans les processeurs multi-coeurs car il implémente des stratégies à écriture différée, minimisant le trafic mémoire.
Nous présentons le protocole *MOESI* à titre d'exemple. *MOESI* permet à une ligne de cache d'avoir cinq états différents. Le passage entre les différents états est résumé dans la figure A.35. Chaque coeur surveille toutes les commandes effectuées sur le bus pour mettre à jour l'état de ses lignes ou les communiquer quand il en est propriétaire.

L'état M (*modified)* indique que la ligne est valide et qu'elle a été modifiée dans ce niveau de cache et qu'elle est seulement présente dans ce cache. La valeur en mémoire n'est pas cohérente, la ligne doit alors être copiée en mémoire lors de son remplacement.

L'état O (*owned* ou *shared-modified*) indique que cette ligne est valide est qu'elle est présente dans au moins un autre niveau de cache. Le cache actuel est *propriétaire* de cette ligne, il doit informer les autres caches lors de sa modification. La ligne modifiée peut ensuite être communiquée à un autre niveau de cache, sans avoir à passer par la mémoire. Cet état est la principale amélioration apportée par le protocole *MOESI* au protocole *MESI*.

L'état E (*exclusive*) indique que la ligne est valide uniquement dans ce niveau de cache. Cela évite l'émission d'invalidation aux autres caches qui ne détiennent pas cette ligne. De plus, sur un autre cache souhaitant accéder à cette donnée, la ligne de cache peut directement être transférée depuis le cache sans accès mémoire. La ligne dans le premier cache passera alors de l'état E à O. Dans le deuxième cache la ligne sera en état S.

L'état S (*shared*) indique que la ligne est valide dans le cache courant et dans au moins un autre cache. Le cache actuel n'est pas propriétaire de la ligne (état O). La cohérence avec la mémoire n'est pas assumée.

L'état I (*invalid*) indique que la ligne n'est pas valide. La lecture de cette ligne est interdite.

Protocole globaux - Cohérence par répertoire (*directory based coherence*). La seconde famille regroupe les protocoles dits *globaux* utilisent des répertoires et des contrôleurs émettant les commandes de transferts des lignes de cache (entre les caches ou avec la mémoire)[Tan76]. Toutes les informations nécessaires à la gestion de la cohérence sont enregistrées dans un répertoire. Leur performance est meilleure que les protocoles utilisant des techniques de *snooping* et *broadcasting* car ils génèrent moins de trafic. Bien que les protocoles tels que MOESI réduisent le trafic mémoire en utilisant des écritures différées, la gestion des cinq états est complexe. Et le trafic généré par la cohérence de cache augmente fortement avec le nombre de coeurs utilisés et peut rapidement voir ses performances s'effondrer [Liu16]. Les futures architectures à mémoire partagée nécessiteront d'implémenter des protocoles de cohérence de cache très performants [AlH+10].

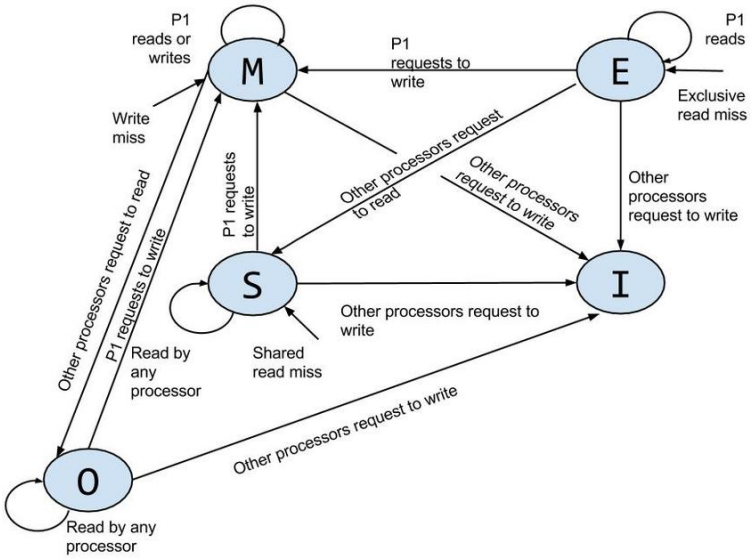


FIGURE A.35 – Fonctionnement du protocole MOESI (source [SMA14])

A.4 Mémoire virtuelle

La section précédente explique pourquoi la mémoire est une des ressources les plus importantes des architectures modernes. Sa gestion doit être la plus performante possible si l'on souhaite minimiser au maximum le trou de performance séparant les mémoires et les processeurs. Si la hiérarchie de mémoire est la réponse matérielle à ce challenge la mémoire virtuelle est une réponse logicielle. Avant de la présenter en détail, cette introduction a pour but de motiver son utilité.

Utilité de l'abstraction de la mémoire. Sans abstraction mémoire, tous les programmes et le système d'exploitation partageraient le même espace d'adressage. Cette implémentation, utilisée par les premières architectures, a deux inconvénients majeurs. Le premier concerne la sécurité de l'exécution d'un programme. S'il venait à écrire dans une zone mémoire réservée au système d'exploitation, un arrêt brutal du système pourrait survenir. De plus, lors de l'exécution de plusieurs processus sur le même processeur, deux programmes différents pourraient accéder ou modifier des données ne lui appartenant pas. Une solution pour contourner ce problème est d'alterner l'exécution de chaque processus en vidant et chargeant ses données depuis le stockage, engendrant le deuxième inconvénient d'un système sans abstraction mémoire : la performance. Bien que des threads puissent tout de même être utilisés (ils appartiennent au même processus et ont accès au même espace mémoire) l'utilisation de cette architecture serait très impactée. Par exemple, un utilisateur ne pourrait pas avoir plusieurs fenêtres exécutant des programmes différents en parallèle. L'utilisation de serveurs multi-utilisateurs ne serait alors même pas envisageable. L'absence d'abstraction mémoire, ou adresse directe, ne trouve d'application aujourd'hui que dans les systèmes embarqués. Le constructeur du système est

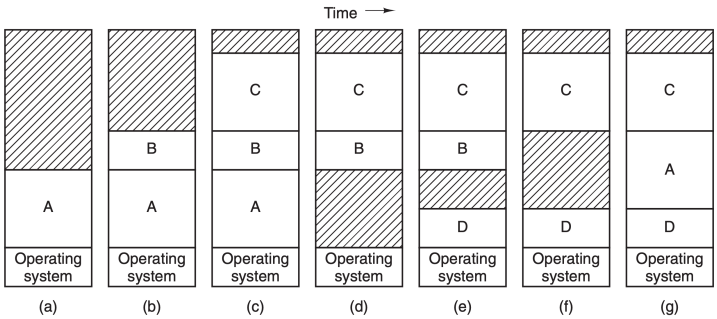


FIGURE A.36 – Technique de va-et-vient pour gérer la mémoire dynamiquement. Les processus A, B, et C sont créés dans les étapes (a), (b) et (c). Lors de la création d'un processus D à l'étape (d), le système d'exploitation doit enlever un processus de la mémoire pour lui faire de la place. Lors de l'étape (f) et (g), le processus B laisse sa place pour que A puisse continuer son exécution. Entre l'étape (c) et (g) le processus A est exécuté à partir de deux espaces d'adressage physique différents (graphique extrait de [Tan08]).

généralement le seul utilisateur du processeur et est donc maître de son utilisation et peut réaliser des allocations mémoires manuellement.

L'abstraction par réallocation statique a été implémentée sur l'ordinateur IBM 260 en 1965 [Bri] pour permettre l'exécution simultanée de plusieurs processus. Le système d'exploitation alloue une adresse de base à chaque processus. Lorsqu'un processus réalise un accès mémoire, un matériel s'occupait de décaler tous ses accès mémoires à partir de l'adresse de base. Ce mécanisme, invisible pour le programmeur était fonctionnelle, mais impactait les performances du programme. Elle pouvait s'avérer complexe à mettre en place, car il fallait distinguer les adresses à convertir et celle ne le nécessitant pas (un saut en mémoire par exemple).

L'abstraction de l'espace d'adressage permet de donner son propre espace d'adresses à chaque processus indépendant les uns des autres. L'allocation dynamique permet de mapper l'espace mémoire d'un processus à un espace physique de la mémoire en utilisant deux registres *base* et *limite* comme sur le processeur Intel 8088. La méthode de *va-et-vient* ou *swapping* peut être utilisée pour gérer les déplacements des processus entre la mémoire et le stockage. Cette méthode est illustrée dans la figure A.36. L'inconvénient de cette méthode est la création de trous dans la mémoire, empêchant son utilisation optimale. Des techniques de compactage ont été alors élaborées, mais étaient souvent très coûteuses (5s pour compacter 1GB de mémoire [Tan08]). De plus cette méthode ne permet pas de gérer les grands logiciels dont la taille ne permet pas d'être stockés en intégralité. Bien que des techniques utilisant les segments de recouvrement (*overlays*) [SW92] aient permis d'adapter le *va-et-vient* à ces grands processus, la technique adoptée depuis est connue sous le terme de *mémoire virtuelle*.

A.4.1 La pagination

La mémoire virtuelle a été implémentée pour gérer de façon efficace des processus dont la taille est plus grande que l'espace mémoire disponible. L'optimisation par *overlay* présentée précédemment était très compliquée à mettre en oeuvre et devait être réalisée par le programmeur. La seconde motivation était de gérer efficacement la mémoire lorsque la somme des tailles des processus exécutés dépasse l'espace mémoire disponible. En d'autres termes, il fallait un mécanisme permettant l'exécution d'un programme sans qu'il soit chargé en totalité en mémoire. La solution devait aussi permettre de gérer facilement les changements de taille des processus de façon efficace, sans avoir à recopier la totalité du programme lors d'une allocation mémoire (*malloc*). Enfin, la mémoire virtuelle doit assurer la sécurité de l'exécution de plusieurs programmes sur une même architecture en évitant les bugs et les vols de données.

A.4.1.1 Les pages

Le principe de la mémoire virtuelle repose sur le principe de donner à chaque processus sont propre espace d'adressage mémoire. Chaque processus peut travailler sur l'adresse *0x100*, car en réalité le mécanisme de mémoire virtuelle fait correspondre cette **adresse virtuelle** à différentes **adresses physiques**. Pour cela, son **espace d'adressage virtuel** est découpé en petites entités appelées **pages** qui contiennent un **espace d'adressage physique** contigu. Chaque page est *mappée* sur des adresses physiques (aussi contiguës) formant un **cadre de page** (*page frame*). Une page et son cadre de page associé contiennent le même nombre d'adresses, dont la taille est choisie par le système d'exploitation. Deux pages contiguës ne correspondent pas forcément à deux cadres de pages contiguës. Ces concepts sont résumés dans la figure A.37. La page 2 contient les adresses virtuelles allant de l'adresse 0 à l'adresse 4095. Lorsque le processus propriétaire de cette page réalise un accès à cette adresse virtuelle, il réalise, sans le savoir, un accès aux adresses physiques se trouvant entre 8192 et 12287. Ni la mémoire ni le processeur n'ont connaissance de cette traduction qui est réalisée par un module matériel indépendant appelé *Memory Management Unit* (MMU) (voir section A.4.2).

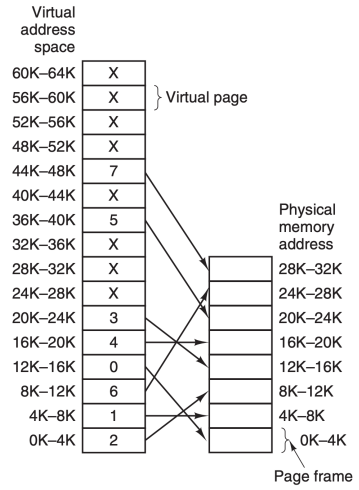


FIGURE A.37 – Correspondance entre les adresses virtuelles, stockées dans des pages, et les adresses physiques, stockées dans des cadres de pages. [Tan08])

.

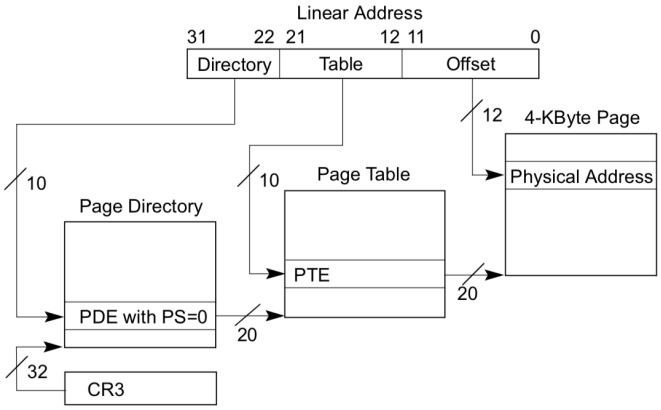


FIGURE A.38 – Table de pages utilisant 32 bits de l'adresse virtuelle dans une table à 3 niveaux [Int11]

.

A.4.1.2 La taille des pages

Les transferts de données entre la mémoire et le stockage se font par page. Ainsi une page ne peut se trouver à la fois en mémoire et sur le disque. En fonction des applications et de l'algorithme de remplacement de pages (voir section A.4.2.2) ces transferts peuvent être fréquents. Le choix de la taille de page doit alors être pris en considération pour obtenir les performances de l'application attendues. Plus la taille des pages est petite, plus l'utilisation effective de la mémoire sera proche de la quantité mémoire disponible. Avec de grandes pages, les processus n'en utilisant qu'une faible partie réduisent la mémoire disponible pour les autres processus. Si

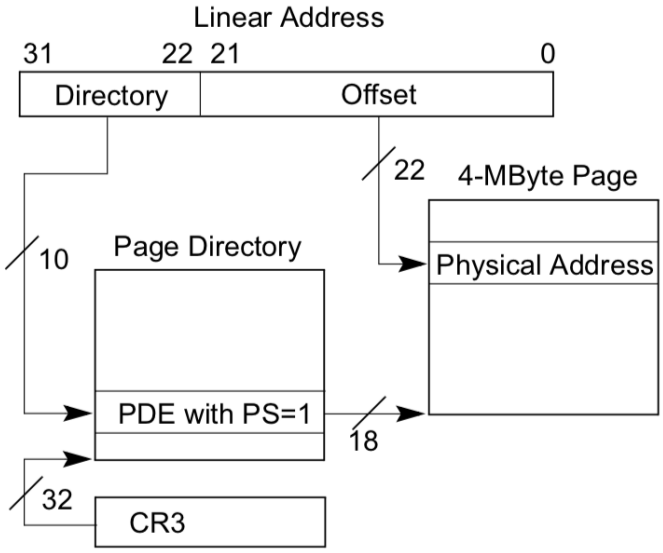


FIGURE A.39 – Table de pages pour des pages larges de 2 MiB[Int11]

.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

FIGURE A.40 – Structure d'une entrée dans la table de page [Int11]

.

les *défauts de pages* sont fréquents, la quantité de mémoire à déplacer entre la mémoire et le stockage et d'autant plus grande.

Aujourd'hui, les systèmes d'exploitation utilisent des pages mesurant $4KiB$. Cette taille est un bon compromis entre la gestion complexe de la table des pages qui doit être parcourue le moins souvent possible, et la meilleure gestion de la mémoire possible. Cependant, les systèmes d'exploitation récents permettent d'utiliser des tailles de pages plus grandes pour certaines applications qui pourraient en bénéficier. Ces grandes pages ou *large pages* ou *huge pages*, sont des pages de taille allant de 2 MiB à plusieurs GiB et peuvent être allouées de deux façons. La première est transparente pour l'utilisation. C'est le système d'exploitation qui analyse les accès et *comprend* que le jeu de données accédées est grand et que l'application pourrait profiter de l'utilisation de grande page. Ce mécanisme est appelé *Transparent Huge Pages* (THP), car il est géré automatiquement par le système [Lin19c]. La deuxième façon pour utiliser des grandes pages est d'allouer la mémoire manuellement dans le code. Un exemple d'allocation est donné dans le code du noyau Linux [Lin19a].

Avec des pages larges, la TLB (voir section A.4.2.1) est plus rapide à parcourir, sa latence est donc réduite. De plus, leur utilisation vise à réduire le nombre de *fautes de page*, les pages couvrant un espace d'adressage plus large. De plus, les pages étant plus grandes, les cadres de pages correspondant le sont aussi : les adresses mémoires sont contiguës sur un plus grand intervalle d'adresse. Cela peut avoir pour effet de réduire les conflits d'associativité dans les caches. Notamment pour des tailles de cache proche de la taille d'une ligne de cache [Lin19b].

A.4.2 Memory Management Unit (MMU)

La *MMU* est un composant matériel responsable de la gestion de la mémoire paginée. Il est, depuis le processeur 80386 d'Intel, intégré directement au processeur. Les missions de la MMU sont multiples. Elle est responsable de la traduction des adresses virtuelles en adresses physiques. Lorsque le processeur réalise un accès mémoire, il n'envoie pas directement l'adresse sur le bus mémoire. Cette adresse (virtuelle) est d'abord traduite (en adresse physique) par la MMU qui s'occupe de l'écrire sur le bus mémoire. La *MMU* doit aussi s'occuper des déplacements des pages entre la mémoire et le disque quand c'est nécessaire (défaut de page). Elle est aussi responsable de sécuriser les accès et d'empêcher un programme d'écrire dans une page qui ne lui appartient pas. Il peut alors lever une exception ($SIGSEGV$) pour interrompre le programme.

A.4.2.1 Table des pages

La table des pages a pour fonction de faire correspondre les pages virtuelles (*Virtual Page Number* (VPN)) à leur cadre de page correspondant (*Physical Page Number* (PPN) représenté par les flèches de la figure A.37. C'est grâce à cette table que les adresses virtuelles peuvent être traduites. L'adresse de cette table est stockée dans un registre (*Page Table Base Register* ($PTBR$). Une fois la traduction de l'adresse virtuelle de la page réalisée, les bits de décalage *Virtual Page Offset* (VPO) sont utilisés pour sélectionner la donnée voulue dans cette page. La figure A.41a montre un exemple du fonctionnement de la table des pages pour la traduction

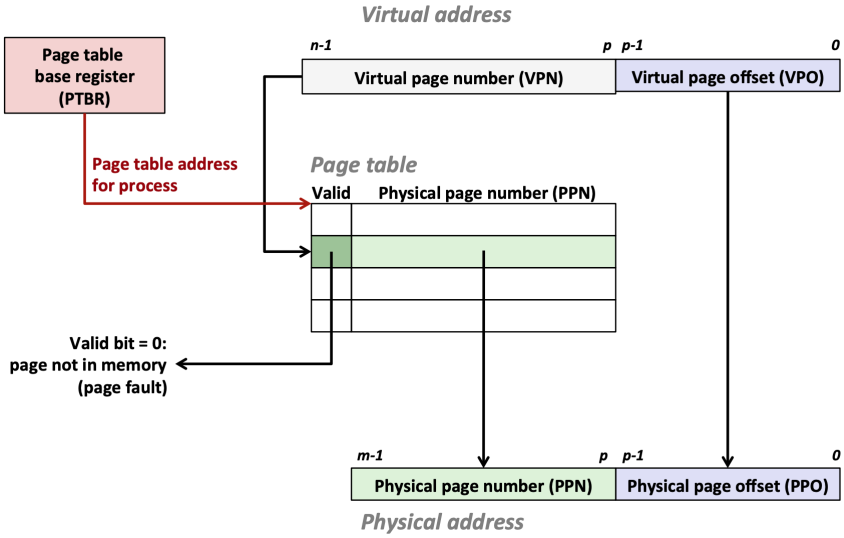
d'une adresse virtuelle. Dans la table des pages est stocké le numéro du cadre de page, utilisé pour la construction de l'adresse physique. Cette entrée contient également d'autres informations (figure A.41b) comme les droits d'accès à la page ou si elle a été modifiée (pour la gestion de cohérence).

Cependant, la table des pages peut rapidement prendre beaucoup de place et son stockage en mémoire peut affecter les performances des programmes. Par exemple, une machine utilisant des adresses de 32 bits aura une table de page mesurant plus de 16 MiB, et chaque processus possède sa propre table. De plus, pour des pages de 4 KiB et un espace d'adressage de 32 bits utilise 1 million de pages. Le parcours de la table peut être long, d'autant que les architectures modernes utilisent des adresses de 64 bits. Pour répondre à ce challenge, les architectes ont implémenté deux techniques permettant d'accélérer la traduction. La première est d'utiliser un cache qui stocke les traductions récentes et évite la traduction permanente des adresses virtuelles. La deuxième est de modifier la structure de la table de pages en implémentant une table à plusieurs niveaux grâce à une structure d'arbre.

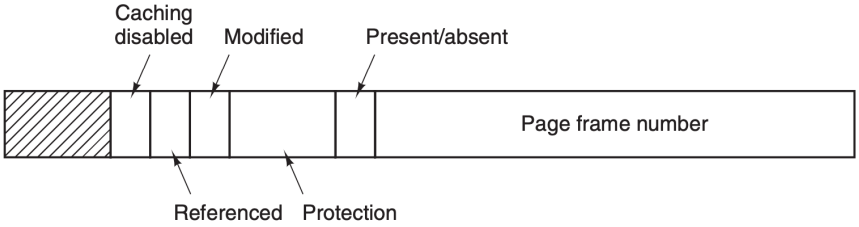
Accélérer le parcours de la table grâce à une table multiniveaux Le parcours de la table de page est primordiale, car une seule instruction peut nécessiter de la parcourir 3 fois avant d'être exécutée (adresse de l'instruction, et adresse des deux opérandes). La figure A.42 montre une table utilisant 4 niveaux. Le premier niveau est appelé répertoire des pages (*Page Directory*). La MMU partage l'adresse virtuelle de la page (VPN) en k morceaux permettant de parcourir les k niveaux de la table de pages. Les derniers bits sont utilisés pour le décalage dans la page mémoire et la construction de l'adresse physique. Ainsi les différentes pages n'ont pas besoin d'être stockées continuellement en mémoire. Seul le répertoire des pages, entrée commune à toutes les pages, est conservé en mémoire.

Le cache de traduction : Translation Lookaside Buffer. Les pages étant de plus en plus grandes (4 KiB sur les architectures actuelles), les processus font généralement beaucoup d'accès à des adresses appartenant à un nombre de pages réduit. Cette observation est à l'origine de la création d'un cache **TLB** conservant les traductions de pages récentes. Ce dispositif matériel est généralement disposé dans la MMU. Dans ce cache sont généralement stockés : l'adresse de la page et du cadre de page correspondant, les droits et protections, la validité ainsi qu'un bit de modification (*dirty bit*). La figure A.43 résume comment le TLB est utilisé par la MMU lors de la traduction d'une adresse virtuelle. Deux cas sont possibles : la page est présente dans le TLB (évènement *TLB-hit*, figure A.43a), ou elle ne l'est pas (évènement *TLB-miss*, figure A.43b).

Pour une architecture récente telle que Skylake, le cache de traduction est composé de trois caches répartis en deux niveaux. Deux caches de niveau 1 se partagent les adresses d'instructions ($ITLB$) et de données ($DTLB$) et possèdent des entrées pour les différentes tailles de pages (4 KiB, 2 MiB, 4 MiB, 1 GiB). Le niveau 2 unifie les deux caches de niveau 1 grâce un cache utilisant 12 sets (*12-way set associative*) et 1536 entrées [Wik].



(a) L'adresse physique de la table des pages est stockée dans un registre ($PTBR$). La MMU utilise une partie des bits de l'adresse virtuelle (VPN) pour trouver la ligne correspondant à la page dans cette table. A cette entrée de la table est stockée l'adresse du cadre de page correspondant à la page virtuelle (PPN). L'adresse PPN associée aux bits restant de l'adresse virtuelle (VPO) permet de construire l'adresse physique de la donnée voulue (graphique tiré de [MR12]).



(b) Pour le fonctionnement de la MMU, la table des pages conserve plusieurs informations importantes pour chaque entrée de page, en plus de l'adresse du cadre de page. Un bit de présence permet de savoir si la page est en mémoire, ou sur le disque. La protection assure qu'un utilisateur ne puisse pas modifier des données ne lui appartenant pas. Un bit de modification est utilisé pour la cohérence [Tan08].

FIGURE A.41 – Exemple de fonctionnement de la MMU pour la traduction d'une adresse virtuelle en utilisant une table de page à un seul niveau.).

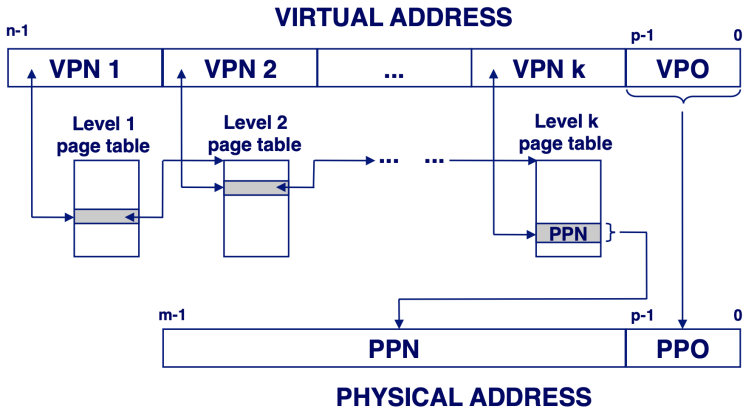
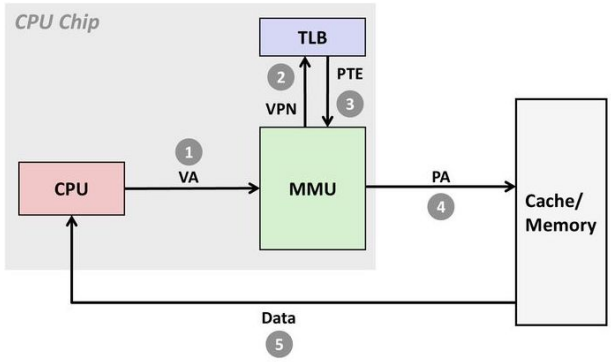


FIGURE A.42 – Exemple d'utilisation d'une table de pages à k niveaux [MR12]. L'adresse virtuelle est découpée en $k+1$ morceaux. Le premier morceau (*VPN 1*) détermine la ligne dans la première table (située à l'adresse *PTBR*). Cette entrée permet d'obtenir l'adresse physique de la page de niveau 2. L'entrée dans cette page est déterminée à partir du second morceau (*VPN 2*). Ce même procédé est utilisé pour parcourir tous les niveaux table. Dans la dernière table (*VPN k*) est stockée l'adresse physique de la page ciblée (*PPN*). L'adresse physique est construite à partir de cette adresse *PPN* et du décalage *VPO* (derniers bits de l'adresse virtuelle).

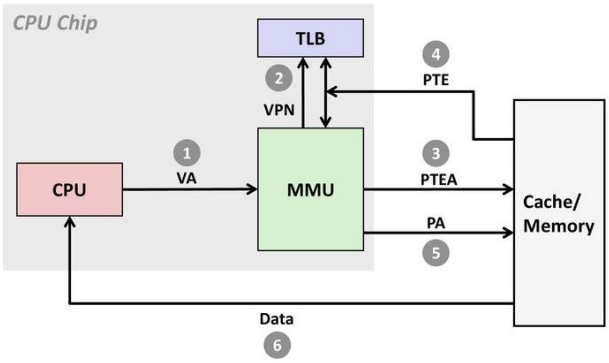
.

A.4.2.2 Déplacement de page

Lorsque la demande mémoire est supérieure à l'espace disponible, la totalité des processus ne peut pas y être stockée. Ainsi, la totalité des pages allouées ne se trouve pas toute en mémoire à un instant donné. Pour cela, la MMU tient une liste des pages se trouvant en mémoire en utilisant pour chaque page un bit de présence/absence. Dans l'exemple de la figure A.37, si un processus accède à l'adresse 33000, sa table des pages et constate que la page n'est pas présente en mémoire. Cet évènement est appelé un **défaut de page** (*page fault*). Le système doit alors déplacer une page de la mémoire vers le stockage pour faire de la place pour cette nouvelle page. Le choix de la page à déplacer se fait grâce à un algorithme de remplacement de page. De la même façon que pour la gestion des lignes de cache, il faut une méthode efficace de remplacement pour éviter de remplacer une page qui sera accédée par l'instruction suivante. Plusieurs algorithmes existent : First In First Out (FIFO) remplace la page la plus vieille, Not Recently Used (NRU) remplace la page non utilisée depuis longtemps, Seconde Chance implémente l'algorithme FIFO, mais cherche en priorité une page non référencée. Pour des applications réalisant des accès mémoire sur plusieurs pages, l'algorithme choisi peut avoir un fort impact sur sa performance.



(a) Si la traduction de la page se trouve dans la TLB, elle renvoie le pointeur vers l'entrée de la page (PTE) directement à la MMU (étape 3) qui peut alors réaliser la traduction de l'adresse virtuelle. Elle accède ensuite directement à la donnée voulue en mémoire grâce à l'adresse physique traduite *PA* (étape 4)



(b) Si la page ne se trouve pas dans la TLB, la MMU doit s'occuper de la traduction de PTE. Elle utilise sa table des pages (multiniveaux par exemple) pour trouver l'adresse physique *PTEA* où est stockée l'adresse de l'entrée de la page *PTE* (étape 3). La traduction est copiée dans la TLB pour éviter une nouvelle traduction (étape 5). Enfin, elle peut réaliser la traduction de l'adresse virtuelle en adresse physique (étape 5)

FIGURE A.43 – Fonctionnement de la TLB. Lors d'un accès mémoire le processeur envoie l'adresse virtuelle *VA* à la MMU pour la traduire (étape 1). La MMU vérifie si la traduction *VPN* a été faite récemment et si l'adresse physique a été sauvée dans la TLB (étape 2) [MR12].

Annexe B

Les compteurs matériels

Sommaire

B.1 Les compteurs **283**
B.1.1 Les compteurs matériels 283
B.1.2 Mode de mesure : comptage et échantillonnage 285
B.1.3 Compteurs matériels des architectures Intel 286
B.2 Unité de suivi de performance Intel : les PMU **289**
B.2.1 PMU core 290
B.2.2 PMU uncore 292
B.3 Discussion **297**
B.4 Conclusion **299**

Les compteurs matériels (*hardware counters*) sont des registres présents sur les processeurs permettant de compter les événements (matériels ou logiciels) avec un impact minimal sur la performance du code exécuté. Ces compteurs sont aussi présents sur d'autres composants du système, tels que les contrôleurs de mémoire et les interfaces réseau (voir figure B.1). Les données récoltées peuvent être utilisées pour l'évaluation des performances et des réglages (BIOS, système d'exploitation). Cette annexe s'intéresse à l'évolution des compteurs matériels sur les architectures Intel. Elle complète leur présentation réalisée dans le manuscrit, à la section 2.4. Nous présentons comment ces derniers sont implémentés sur les architectures récentes du constructeur et quels sont les moyens à la disposition des programmeurs pour les utiliser.

B.1 Les compteurs

B.1.1 Les compteurs matériels

Pour pouvoir suivre l'activité d'un processeur lors de son fonctionnement, les architectures se sont dotées de matériels spécifiques appelés compteurs matériels (*hardware counters*). À l'origine, ce dispositif était seulement utilisé par les constructeurs des puces à des fins de débogage lors de la conception d'une nouvelle microarchitecture. Avec le processeur 80386, Intel a introduit deux registres, nommés `TR6` et `TR7`, utilisés pour valider la performance du tampon d'anticipation de conversion (*Translation Look-aside Buffer* (TLB)). Intel précisa alors que ces deux registres ne seraient présents que sur cette version du processeur. Cependant Intel conserva

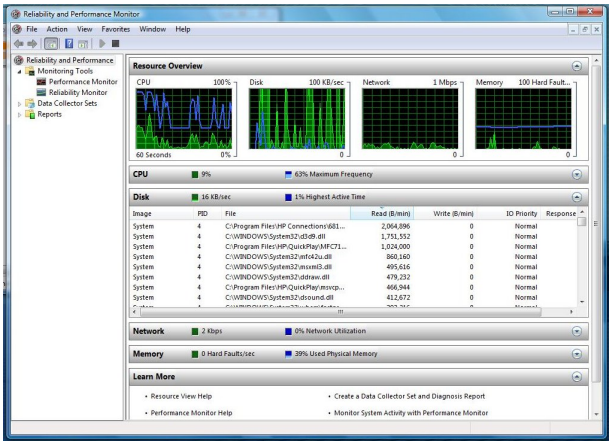


FIGURE B.1 – L'interface de suivi de performance de Windows permet de suivre l'activité de différentes ressources d'un ordinateur : processeur, mémoire, stockage, réseau...

ces deux registres et en ajouta trois dans la version suivante du processeur, le 80486, tout en répétant que ces registres pourraient disparaître lors de nouvelles versions d'architectures. Cependant, les utilisateurs se sont approprié ces compteurs pour développer leurs propres outils d'analyse de performance. Avec la génération de processeur suivante (80586), Intel retira les compteurs matériels présents dans la version précédente, rendant les outils de suivi de performance inutilisables. Cependant, avec le développement du processeur Pentium, Intel a fourni aux développeurs un moyen d'accéder de manière uniforme aux différents compteurs : les *Model Specific Registers* (`MSR`). Ces registres sont séparés en deux familles : les registres architecturaux et non architecturaux :

— **Les registres architecturaux** sont des registres qui ne "*devraient*" [1] pas disparaître d'une version à l'autre d'un processeur. Pour des raisons historiques, ces registres sont référencés depuis le Pentium 4 avec le préfixe `IA32_`. En plus de ne pas disparaître avec les nouvelles architectures, ces registres ne varient pas non plus entre différentes lignes de produits. Par exemple, un registre architectural peut être utilisé sur un processeur de téléphone comme sur un processeur de serveur. Par exemple, le registre `IA32_TIME_STAMP_COUNTER` est localisé à l'adresse `0x10` depuis le processeur Pentium. Ce compteur est incrémenté à chaque cycle d'horloge (de base). La fréquence d'incrémentation de ce registre ne varie pas avec celle du processeur et permet de mesurer très précisément un intervalle de temps entre deux mesures.

— **Les registres non architecturaux** quant à eux ne sont pas assurés d'être présents sur de futures architectures. Par exemple, le registre utilisé pour mesurer le nombre d'opérations à virgule flottante (FLOP) exécutées, a été retiré entre les architectures Sandy Bridge et Haswell avant de réapparaître ensuite.

1. Documentation Intel [Int18] - *The MSRs in the Pentium processor are not guaranteed to be duplicated or provided in the next generation IA-32 processors*

B.1.2 Mode de mesure : comptage et échantillonnage

Qu'ils soient architecturaux ou non, le but des `MSR` est de permettre à l'utilisateur de mesurer le nombre d'occurrences d'un évènement. Cet évènement peut être matériel (cycle d'horloge, exécution d'une instruction, manque dans un cache...) ou logiciel (faute de page, changement de contexte...). Comme les évènements dépendent de la présence ou non des compteurs correspondants, la pérennité des évènements n'est pas assurée d'une architecture à l'autre. Lors du développement d'un outil d'analyse de performance, il est nécessaire de vérifier qu'ils sont disponibles avec l'architecture utilisée. Intel propose la liste des évènements compatible avec chaque architecture [2]. Il existe deux méthodes permettant de suivre le nombre d'évènements ayant lieu sur la microarchitecture : le comptage et l'échantillonnage.

Le comptage. La première méthode consiste à compter le nombre d'évènements arrivant entre deux intervalles de temps. Pour cela, le compteur est initialisé à 0 et lu au bout d'une certaine période de temps. Les valeurs ainsi récupérées permettent de mesurer le nombre d'occurrences de ces évènements. Cette méthode est efficace, mais elle ne permet pas de connaître la partie du code responsable d'un évènement. Les ratios d'évènements, tels que les instructions exécutées par cycle, les taux de *miss* de mémoire cache et les taux d'erreurs de prévision des branches peuvent ensuite être calculés.

L'échantillonnage. Pour obtenir plus d'informations sur les instructions responsables du déclenchement d'un évènement, le mode d'échantillonnage (*sampling*) doit être utilisé. Ce mode consiste à déclencher une interruption tous les n évènements et sauvegarder certaines informations telles que la valeur du pointeur d'instruction. Pour cela, le registre de comptage est initialisé à la valeur $\text{MAX} - n$ où MAX correspond à la valeur maximale pouvant être stockée dans le registre. Lorsque n évènements sont comptés, le registre déclenche un débordement (*overflow*) et génère une exception traitée par le système d'exploitation. Grâce à un échantillonnage assez fin (nombre n petit) et des méthodes de statistiques, il est possible d'approcher le nombre d'évènements généré par chaque instruction. La principale difficulté de cette technique est d'assurer suffisamment de précision lors de l'attribution d'un évènement à une instruction. En effet, entre le moment où l'interruption est générée et son traitement, plusieurs instructions peuvent avoir été exécutées. Des technologies telles que Intel PEBS [3] (Processor Event-Based Sampling) ou AMD IBS (Instruction Based Sampling) [Dro07] agrémentent le processeur d'un tampon lui permettant de stocker les informations nécessaires. L'autre avantage de cette technologie est de réduire l'impact sur les performances dû au traitement de chaque échantillon par le système d'exploitation. La PMU possède un tampon pouvant stocker plusieurs échantillons et n'interrompt l'exécution que lorsque ce tampon est plein. Le principal désavantage de cette technologie est le nombre restreint d'évènements compatibles. De plus, elle n'est pas compatible avec toutes les architectures réduisant la portabilité des outils l'utilisant.

2. Liste des évènements compatibles par architecture Intel - https://download.01.org/perfmon/index/
3. Documentation Intel - Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, Chapter 18. https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf

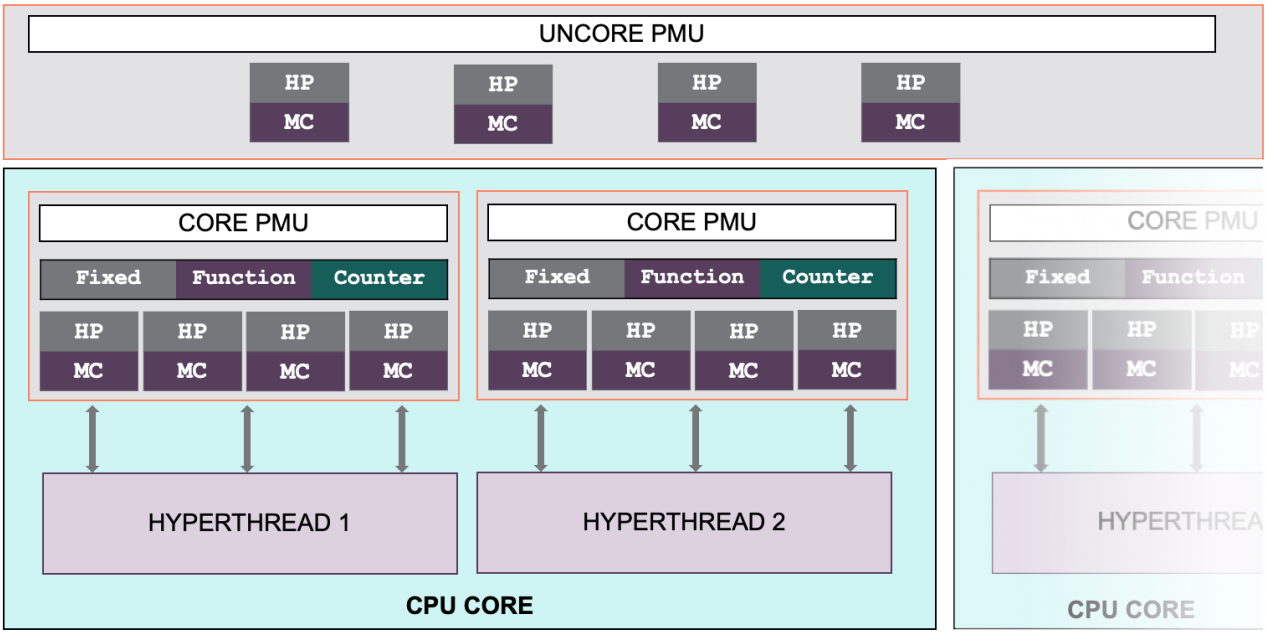


FIGURE B.2 – Disposition des compteurs matériels sur un coeur d'un processeur Intel de génération Skylake. Chaque coeur possède une PMU *oncore* accueillant 3 compteurs FFC et 4 compteurs HPMC. Une PMU *uncore* est situé sur le processeur lui même et possède 4 compteurs HPMC.

B.1.3 Compteurs matériels des architectures Intel

Pour compter le nombre d'occurrences d'un évènement, les processeurs Intel possèdent deux types de compteurs :

— Les *Fixed-function counters* (**FFC**) : compteurs préconfigurés pour compter un évènement, prêts à l'emploi.

— Les compteurs de performance (**HPMC**) : compteurs pouvant être configurés pour compter différents évènements.

Ces compteurs sont situés sur un matériel dédié appelé Performance Monitoring Unit (PMU). Les PMU sont présentées en détail dans la section B.2. Les processeurs Intel possèdent une PMU *uncore* directement sur le processeur et une PMU par coeur logique, donc deux PMU par coeur physique lorsque l'*hyperthreading* est activé (voir figure B.2). Chaque PMU *oncore* possède 3 compteurs *Fixed-Function* et 4 compteurs HPMC.

B.1.3.1 Les compteurs FFC

Les compteurs *Fixed-Function* sont les plus simples à utiliser, car ils sont déjà configurés pour compter un évènement. Sur les architectures Intel, il y a 3 compteurs FFC par coeur logique :

— `IA32_FIXED_CTR0` : compte le nombre d'instructions exécutées par un coeur.

— `IA32_FIXED_CTR1` : compte le nombre de cycles à la fréquence nominale du processeur.
— `IA32_FIXED_CTR2` : compte le nombre de cycles durant lesquels le coeur est **actif**.

La configuration et l'activation de chacun des trois compteurs FFC sont présentées sur la figure B.3. Celle-ci fait appel aux trois `MSR` suivants :

— `IA32_PERF_GLOBAL_CTRL` : ce `MSR` permet d'activer les deux types de compteurs (HPMC et PPF). Chaque bit de ce registre correspond à l'activation ou non du compteur associé. Cette activation n'a à être réalisée qu'une seule fois au démarrage du processeur.
— `IA32_FIXED_CTR_CTRL` : ce `MSR` permet de configurer le comptage d'événements : mode utilisateur ou noyau, mesure du coeur logique associé ou de tout le coeur physique, ainsi que de permettre au compteur de générer ou non une interruption lorsque celle-ci dépasse la valeur maximale pouvant être stockée sur 48 bits. Ce dernier bit de configuration peut être utilisé pour vérifier l'utilisation ou non du compteur par un autre programme (qui voudrait être alerté par une interruption).
— `IA32_FIXED_CTR_x` : ce `MSR` reçoit le nombre d'évènements comptabilisé suite à la configuration du compteur FFC correspondant.

Une fois ces deux opérations réalisées (activation et configuration), les évènements sont comptés dans les trois registres correspondants, il ne suffit alors plus que de lire la valeur s'y trouvant. Bien qu'il ne soit pas nombreux, les trois compteurs FFC permettent d'obtenir des informations intéressantes sur l'utilisation du processeur. Par exemple en divisant le troisième compteur par le deuxième ($\frac{IA32\_FIXED\_CTR3}{IA32\_FIXED\_CTR2}$) on obtient le pourcentage d'utilisation du coeur. En utilisant le premier compteur, on peut aussi obtenir un bon indicateur sur la performance du code en calculant le nombre d'instructions réalisées par cycle d'horloge (IPC).

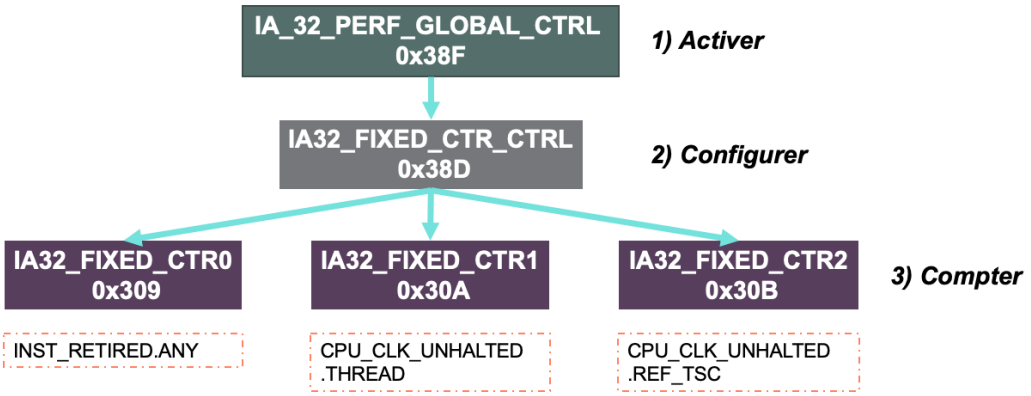


FIGURE B.3 – Chaque coeur logique possède 3 compteurs FFC pouvant être configurés et utilisés à l'aide de trois autres registres `MSR`.

B.1.3.2 Les compteurs HPMC.

Les compteurs HPMC sont plus difficiles à utiliser que les compteurs FFC car ils doivent entièrement être configurés par l'utilisateur et la longue documentation (plusieurs centaines de pages) n'est pas toujours évidente à comprendre. Leur nombre dépend de l'architecture. Les processeurs de génération Skylake possèdent 4 compteurs HPMC par PMU *oncore* (8 compteurs par coeur). Contrairement aux compteurs FFC, les compteurs HPMC sont présents sur les PMU des *oncore* et *uncore*. Un compteur HPMC est composé de deux registres `MSR` présentés sur la figure B.4) :

— Le premier registre est appelé PMC (Performance Monitoring Controler) référencé sous le nom `IA32_PERFEVTSELx`. Chaque compteur HPMC possède son propre registre PMC (accessibles aux adresses 0x186, 0x187, 0x188 et 0x189) qui permet de réaliser la configuration des quatre compteurs : activation, génération d'une interruption, mode (utilisateur ou noyau), évènement à compter (voir figure B.5).

— Le deuxième registre est appelé PMD (Performance Monitoring Data). Il enregistre le nombre d'évènements depuis son activation réalisée avec registre PMC qui lui est associé. Ces `MSR` sont des registres architecturaux nommés `IA32_PMC` et localisés des adresses `0x4C1` à `0x4C4`.

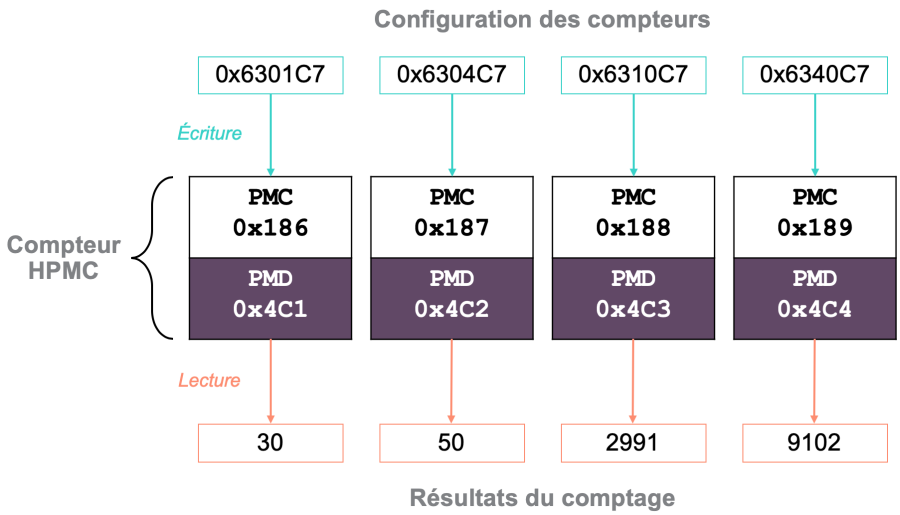


FIGURE B.4 – Chaque coeur logique dispose d'une PMU possédant 4 compteurs HPMC pouvant être configurés pour compter différents évènements. Dans cet exemple, les 4 compteurs sont configurés pour compter toutes les opérations flottantes en double précision exécutées sur le coeur.

Le tableau B.1 montre un exemple de configuration d'un compteur permettant de compter le nombre de FLOP (opérations scalaires et vectorielles (128, 256 et 512 bits)). Il faut pour cela trouver le code correspondant dans la documentation Intel [Int18] au chapitre 19.2 pour les évènements Skylake. Ensuite, les différents bits de configuration sont utilisés pour activer le compteur et filtrer les évènements à compter (système d'exploitation et/ou utilisateur). La

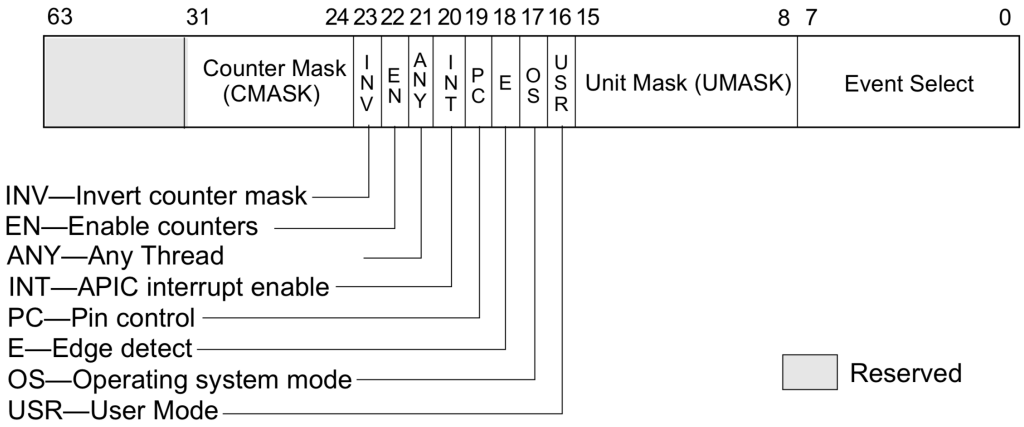


FIGURE B.5 – Organisation du registre `MSR` utilisé pour la configuration d'un compteur.

	CMASK	INV	EN	ANY	INT	PC	E	OS	USR	UMASK	EVENT
bits	31 - 24	23	22	21	20	19	18	17	16	15 - 08	7 - 0
Valeur	0	0	1	1	0	0	0	1	1	1	C7
Résultat	0	6				3				1	C7

Tableau B.1 – Configurer un compteur pour compter le nombre de FLOP scalaires exécutés par un coeur revient à écrire la valeur $0x6301C7$ dans un registre PMC de la PMU du coeur.

valeur résultante (par exemple `0x6301C7` pour les instructions scalaires) n'a plus qu'à être écrite dans un des quatre `MSR` du coeur pour lancer le comptage dans le registre PMD correspondant.

B.2 Unité de suivi de performance Intel : les PMU

Dans la section précédente, nous avons présenté les deux types de compteurs des architectures Intel. Dans cette section, nous présentons les différents moyens à disposition de l'utilisateur pour compter des évènements. Les évènements peuvent être collectés à différents endroits du processeur grâce à un matériel dédié appelé PMU (*Performance Monitoring Unit*). Leur séparation du reste du processeur permet de ne pas impacter la performance de ce dernier, rendant la collecte d'informations non intrusive. Les PMU sont disposées à différents endroits du processeur : certaines sont rattachées à un coeur spécifique, d'autres sur le processeur lui-même (voir figure B.6). Une architecture comme celle des processeurs Skylake possède une PMU par coeur logique et une PMU *uncore*. Pour pouvoir compter des évènements, la PMU doit être configurée à l'aide de registres matériels `MSR`. Les mesures réalisées par la PMU peuvent être filtrées pour mesurer la totalité de l'activité ou seulement celle d'un processus. Ces configurations sont réalisées directement depuis le code ou depuis le système (outils de profilage, noyau). Les PMU peuvent être utilisées de deux façons différentes pour étudier l'activité d'un processeur :

le comptage et l'échantillonnage.

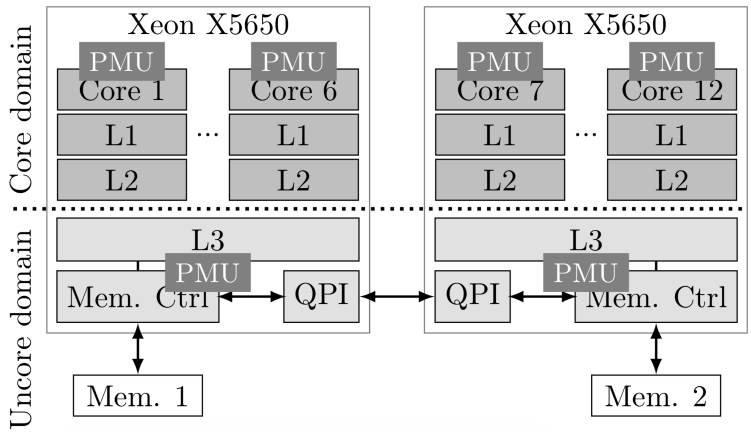


FIGURE B.6 – Exemple de disposition des PMU sur un processeur Skylake (extrait de [SMM17])

Le multiplexage. Le nombre de compteurs HPMC utilisable sur une architecture est bien inférieur au nombre d'évènements supportés par l'architecture. Il faut donc réaliser un choix des évènements à mesurer. De plus, certains évènements ne peuvent pas être mesurés en même temps pour des raisons architecturales. Pour répondre à ces deux besoins, les PMU peuvent utiliser la technique du multiplexage dont le principe est d'exécuter plusieurs groupes de compteurs matériels en suivant un algorithme d'ordonnancement de type *round-robin.* Pour ce faire, un premier groupe d'évènements est mesuré pendant un court intervalle de temps, puis les évènements sont remplacés par ceux du groupe suivant. Ce remplacement est réalisé jusqu'à ce que le comptage d'évènements soit finalement arrêté. Le multiplexage signifie qu'aucun des groupes de compteurs n'a été exécuté sur l'ensemble du code et que l'on ignore quelle fraction du code a été mesurée avec quel groupe. On suppose que la charge de travail est suffisamment uniforme pour que le nombre d'évènements mesurés puisse être étalonné comme si les groupes avaient été exécutés séparément sur le code entier. Cependant, les techniques de reconstruction des profils peuvent être inexactes [Lim+]. De la même façon que pour l'échantillonnage, l'erreur d'estimation augmente avec le nombre d'évènements à mesurer impliquant un partage plus long des PMU. Pour pouvoir utiliser le maximum de compteurs, il est important de noter que la désactivation de l'*hyperthreading* sur les processeurs Intel désactive aussi la moitié des compteurs disponibles (voir figure B.2).

B.2.1 PMU core

Les compteurs sont des registres matériels, ils peuvent donc être écrits et lus comme n'importe quels autres registres. Au fil des générations, les architectures ont reçu plusieurs instructions permettant d'interagir avec les registres `MSR`. Les processeurs Intel possèdent actuellement 5 instructions nécessitant différents privilèges pour les exécuter (voir figure B.7) :

— `wrmsr` et `rdmsr` permettent respectivement d'écrire et de lire les `MSR`.
— `rdpmc` permet de lire le contenu des registres `MSR`
— `rdtsc` et `rdtscp` permettent de lire deux compteurs pré configurés.

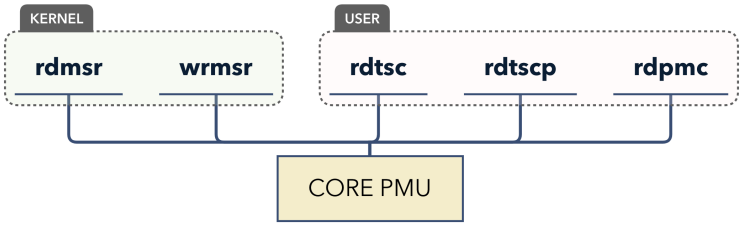


FIGURE B.7 – Instructions assembleurs permettant d'accéder aux registres `MSR` d'une architecture Intel.

Les deux instructions `wrmsr` et `rdmsr` ne sont disponibles qu'en mode *kernel* et ne peuvent être exécutées que par un processus de privilège 0. En effet, laisser la possibilité à un utilisateur normal de modifier ces registres peut être très dangereux pour la sécurité du système. Les trois autres instructions peuvent être exécutées par un utilisateur sans droits supplémentaires. L'instruction assembleur `rdpmc` permet de lire le contenu d'un registre. Cette instruction a été ajoutée avec la sortie du Pentium Pro en 1995. Bien que cette instruction ne permette par d'écrire dans un registre, elle permet de lire le résultat des compteurs et notamment les compteurs FFC. Ces compteurs étant déjà configurés, l'instruction `rdpmc` permet de pouvoir lire ces trois compteurs sans privilège supplémentaire. Dans un programme C, le compteur *ctr* (1, 2 ou 3) peut être lu grâce au code de l'extrait B.1. L'instruction `rdpmc` peut aussi être utilisée pour lire les compteurs HPMC grâce au code de l'extrait B.2. Cela peut être utile pour laisser les utilisateurs accéder aux résultats des compteurs HPMC après qu'un autre programme ayant les droits d'utilisation de l'instruction `wrmsr` ait paramétré les compteurs.

```
1 ul rdpmc_Fixed_Function (int ctr){
2    unsigned a, d, c;
3    c = (1<<30) + ctr;
4    __asm__ volatile("rdpmc" : "=a" (a),
5                               "=d" (d)
6                             :  "c" (c));
7    return ((unsigned long)a) |
8          (((unsigned long)d) << 32);;
9 }
```

Extrait B.1 – Lecture des compteurs FFC grâce à l'instruction `rdpmc`.

```
ul rdpmc_counter (int ctr){
   unsigned a, d, c;
   c = ctr;
   __asm__ volatile("rdpmc" : "=a" (a),
                              "=d" (d)
                            :  "c" (c));
   return ((unsigned long)a) |
          (((unsigned long)d) << 32);;
}
```

Extrait B.2 – Lecture des compteurs HPMC grâce à l'instruction `rdpmc`.

Les instructions `rdmsr` et `wrmsr` sont les instructions bas niveau qui doivent obligatoirement être utilisées pour pouvoir configurer et interagir avec la PMU d'un coeur. Afin de faciliter l'accès et l'utilisation des compteurs, plusieurs interfaces à ces instructions ont été développées (voir figure B.8) et sont présentées dans le manuscrit :

- Perfmon2 (section 2.4.2.3)
- Perf Events (section 2.4.2.3)
- Pseudo système de fichier `/dev/cpu/CPUID/msr` (section 2.4.2.3) ainsi que l'outil Intel `msr-tools`[4]
- PAPI (section 2.4.2.4)
- LIKWID (section 2.4.2.4)

B.2.2 PMU uncore

Une grande partie des CPU modernes se trouve en dehors des coeurs. Sur les processeurs Intel, cette partie est appelée *uncore* et possède entre autres le lien PCI-Express, le cache de dernier niveau ainsi que les contrôleurs mémoire. Pour suivre les performances de ces matériels, l'*uncore* possède également une PMU permettant entre autres de mesurer la bande passante mémoire ou l'activité du cache LLC. La PMU ne se trouvant pas sur les coeurs directement il est impossible d'associer le déclenchement d'un évènement à un coeur particulier et d'autant moins à un processus. De plus, il n'est pas possible de contrôler les registres `MSR` directement depuis le coeur grâce aux instructions assembleurs `rdmsr` et `wrmsr` utilisées pour les PMU des coeurs. Cependant, certains des compteurs *uncore* sont dupliqués sur les coeurs et peuvent être accédé de la même manière (évènement défaut de cache (miss) dans le cache LLC par exemple). Pour accéder aux autres compteurs, il est nécessaire de passer par l'interface PCI.

Rappel PCI. Le bus PCI a été développé pour établir un bus local de haute performance et de faible coût. Le bus PCI et l'interface de carte d'extension sont indépendants du processeur, ce qui permet une transition efficace vers les nouvelles générations de processeurs. Pour identifier les différents matériels PCI, la notation BDF est utilisée : `Bus:Device:Function`. Cela permet d'avoir jusqu'à 256 bus, chacun avec jusqu'à 32 appareils, chacun supportant huit fonctions.

4. https://github.com/intel/msr-tools
5. source : Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual.

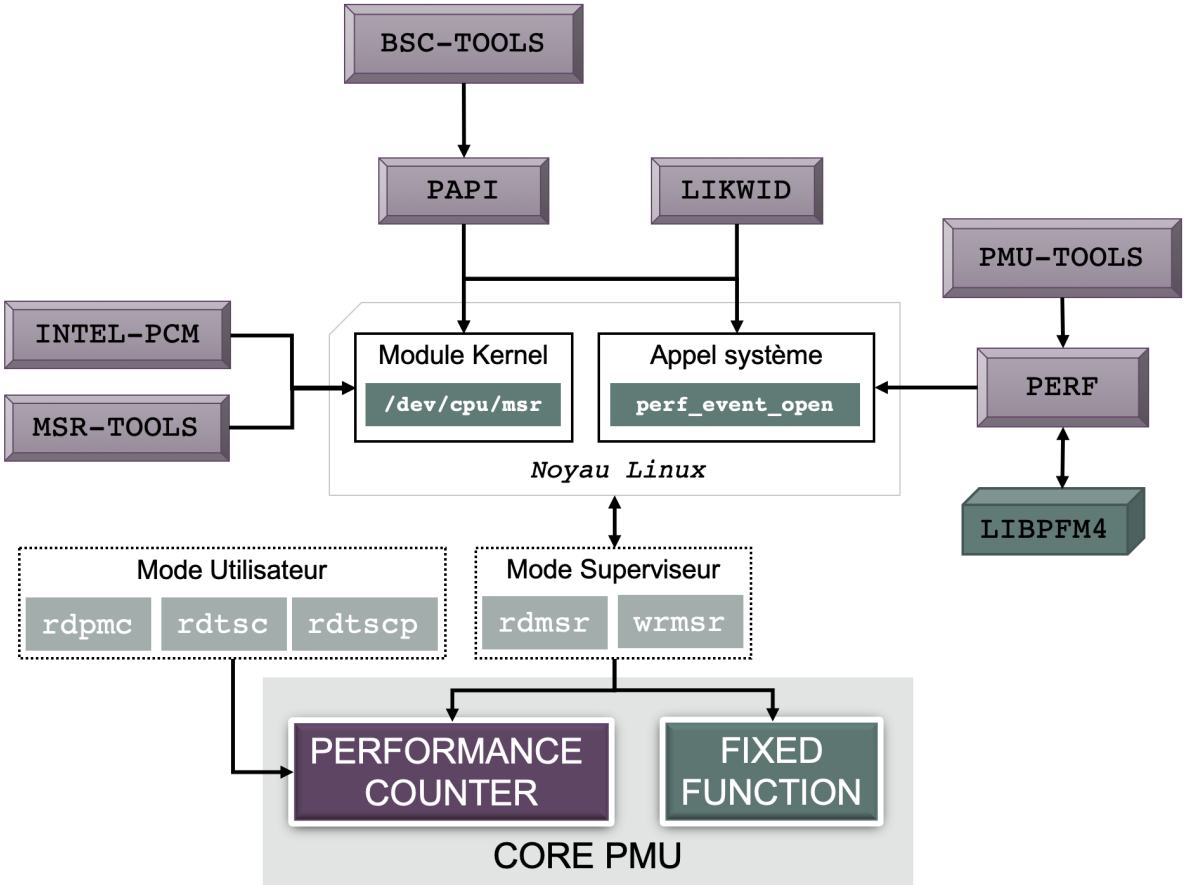


FIGURE B.8 – Différents moyens d'accéder aux compteurs localisés sur la PMU des coeurs. Linux propose deux interfaces permettant d'utiliser les deux instructions `rdmsr` et `wrmsr` : un module noyau et un appel système.

Chaque fonction de l'appareil sur le bus dispose d'un espace de configuration de 256 octets et impose une structure spécifique. Tous les périphériques compatibles PCI doivent prendre en charge les champs| ,`Vendor ID` ,`Device ID` ,`Command and Status` ,`Revision ID` ,`Class Code` et `Header Type`. Par exemple, un matériel construit par Intel aura un champ `Vendor ID` valant 0x8086 [6].

B.2.2.1 /sys interface

L'interface Linux */sys* est un système de fichier virtuel introduit avec la version 2.6 du noyau Linux. Il permet de présenter les différents *devices* du noyau à l'espace utilisateur. Un *device* peut être un matériel, un driver, un bus... Chaque répertoire représente un matériel, et les fichiers s'y trouvant comportent l'information correspondant à son nom (vendeur, évènement). Les matériels peuvent être représentés par bus, grâce au chemin `/sys/bus` comme présenté

6. Source : https://pcisig.com/membership/member-companies?combine=Intel

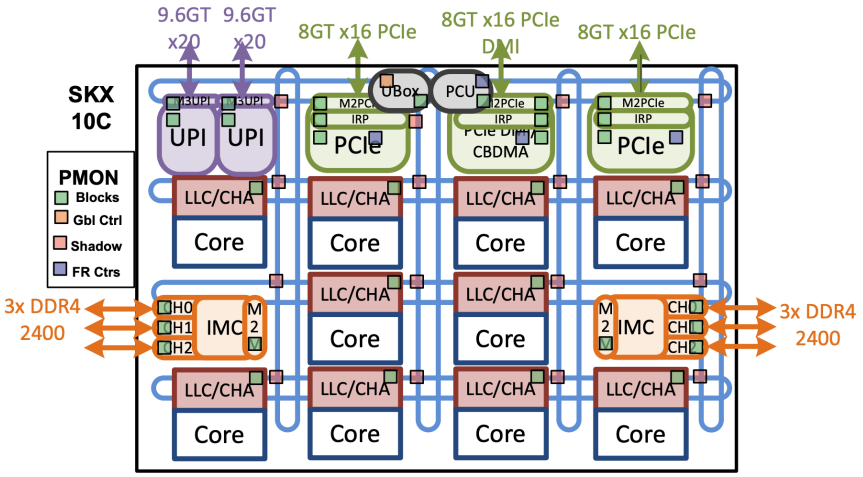


FIGURE B.9 – Disposition des PMU uncore sur les processeurs Xeon Skylake. [5]

dans la figure B.10. Pour accéder aux PMU *uncore*, le driver doit être activé. Il est possible de le vérifier en regardant si plusieurs fichiers nommés *uncore** apparaissent dans le dossier `/sys/devices/`. Il est ensuite possible de programmer les PMU grâce à des lectures/écritures dans le fichier *config* correspondant au bon matériel. L'exemple suivant permet d'activer les compteurs *FFC* :

```
pci_fd = open("/sys/devices/pci0000:3a/0000:3a:0a.6/config",O_RDWR);
ctl = 0x10100UL; // enable + freeze
pwrite (pci_fd, &ctl, sizeof(ctl), BOX_CTL);
```

B.2.2.2 lspci + setpci

Linux propose deux commandes pour interagir avec le matériel PCI. Une commande pour les lister (`lspci`). Une commande pour les configurer (`setpci`) et dans les ténèbres les lier. Cette section présente un exemple de configuration des compteurs permettant de suivre le trafic du bus mémoire d'une architecture Skylake.

Les processeurs Xeon Skylake possèdent deux contrôleurs mémoire (IMC) possédant chacun 3 canaux, pour un total de 6 canaux mémoire. Pour pouvoir configurer ces compteurs, il faut alors obtenir les informations suivantes : identifiant du bus, identifiant du matériel et la fonction. Ces informations peuvent être retrouvées dans la section 1.8.2 de la documentation Intel [Int17] : Identifiant (`0x2042`, `0x2046` et `0x204A`) ainsi que le numéro du bus (2). Pour pouvoir utiliser la notation BDF, il faut alors connaître l'adresse allouée au bus numéro 2 en consultant le registre `0x300` qui stocke l'adresse des différents bus pci pour chaque processeur. La commande `#rdmsr 0x300` renvoie la valeur `800000005d3a1700` et permet de déduire que l'adresse du deuxième bus est `0x3a`. La commande `lspci` permet de s'assurer que les contrôleurs sont bien reconnus et accessibles :

```
lspci | grep "3a"                    //Sélectionner le bus du CPU étudié
```

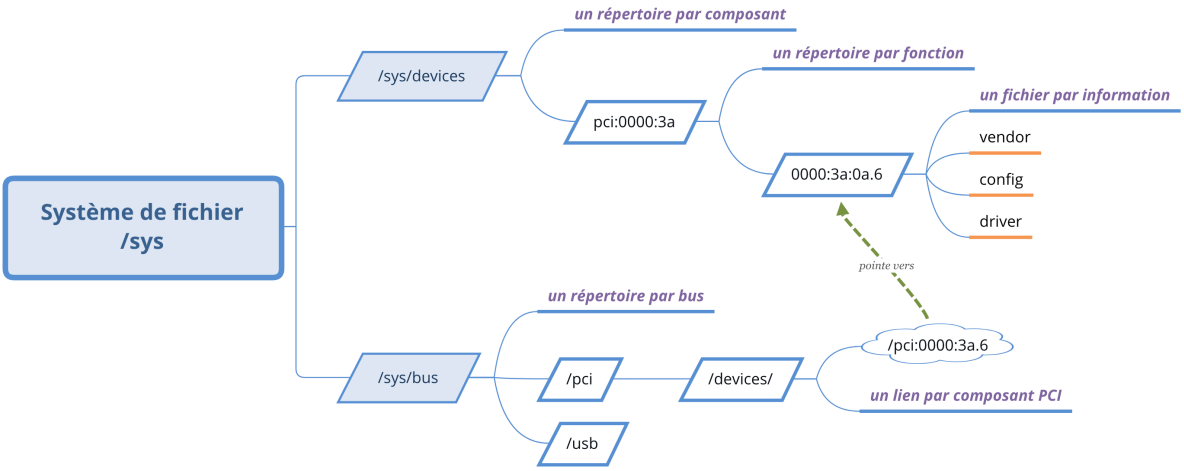


FIGURE B.10 – Système de fichier virtuel présentant les matériels PCI.

```
      | grep -i "\(2042\|2046\|204A\)" //N'afficher que les controleur mémoires
3a:0a.2 System Intel 2042
3a:0a.6 System Intel 2046
3a:0b.2 System Intel 204a
3a:0c.2 System Intel 2042
3a:0c.6 System Intel 2046
3a:0d.2 System Intel 204a
```

À l'aide de la documentation Intel [Int17], il est ensuite possible de configurer les compteurs de chaque canal mémoire grâce au code présenté dans l'extrait B.3. Les évènements comptés sont construits à partir du tableau 2-108 [Int17]. Un script permettant le démarrage, l'arrêt et la lecture des compteurs peut être consulté sur notre dépôt GitHub [7].

7. https://github.com/PourroyJean/performance_modelisation/blob/master/src/tool_PMU/pmu_uncore.sh

```
 Channels=(0 1 2 3 4 5)
      bus=3a
  devices=(0a  0a  0b  0c  0c  0d)
functions=( 2   6   2   2   6   2)

for CHAN in $mychannels ; do
    #       FREEZE and RESET
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmonunitctrl_hi}=0x1
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmonunitctrl_lo}=0x2
    #       ZERO the counter:
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmoncntrupper1_0}=0x0
    #       RESET MONITORING CTRL REGISTERS
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmonunitctrl_lo}=0x1
    #       CONFIGURE COUNTER #0 FOR CAS_COUNT.RD and #1 for CAS_COUNT_WR
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmoncntrcfg_0}=${ev_read}
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmoncntrcfg_1}=${ev_write}
    #       RELEASE COUNTERS
    setpci -s ${BUS}:${devices[CHAN]}.${functions[CHAN]} ${pmonunitctrl_hi}=0x0
    done
```

Extrait B.3 – Configurer les compteurs *uncore* à l'aide de la commande *setpci*.

B.2.2.3 Mapper l'espace d'adressage PCI

La deuxième façon d'accéder aux compteurs matériels *uncore* est de *mapper* l'espace d'adressage PCI dans l'espace utilisateur. Les outils développés chez HPE utilisent principalement cette technique. La première étape est d'obtenir l'adresse du début de cet espace de configuration en utilisant la commande suivante :

```
#cat /proc/iomem | grep MMCONFIG
80000000--8fffffff : PCI MMCONFIG
```

Il est ensuite possible de mapper cet espace mémoire dans l'espace utilisateur. En langage C le code présenté dans l'extrait B.4 permet de réaliser la configuration des compteurs comme dans l'exemple précédemment réalisé avec `setpci`.

```
mymapping =  mmap(0,0x10000000,
                PROT_READ | PROT_WRITE, MAP_SHARED,
                /dev/mem, 80000000);
for(chan=0;chan<nchan;chan++) { //Pour les 6 canaux memoire
  // FREEZE COUNTERS
  PCIDATA8(bus,chan_dev[chan],chan_func[chan],_Config_add_)=0x1;
  // READ COUNTER
  val0 = (PCIDATA32(bus,chan_dev[chan],chan_func[chan], _PMC0_1) & 0x0000ffff )<<32;
  val0|=  PCIDATA32(bus,chan_dev[chan],chan_func[chan], _PMC0_0);
  // ZERO COUNTERS
  PCIDATA32(bus,chan_dev[chan],chan_func[chan], PMC0_1)=0x0;
  PCIDATA32(bus,chan_dev[chan],chan_func[chan], PMC0_0)=0x0;
  ...
}
```

Extrait B.4 – Configurer les compteurs *uncore* en mappant l'espace d'adressage PCI.

B.3 Discussion

Dans cette dernière section, nous discutons de différents aspects concernant l'utilisation des compteurs matériels.

Documentation. La principale difficulté pour l'utilisation des compteurs matériels est leur faible documentation et le manque de connaissance des utilisateurs. Ce manque de documentation à un grand impact sur la quantité et la qualité des outils disponibles. Nous pouvons aussi regretter que ce manque oblige de nombreux programmeurs à pratiquer de la rétro-ingénierie pour comprendre certains comportements de la microarchitecture ou de l'utilisation des compteurs.

Programmer les compteurs sans utiliser les interfaces déjà existantes (PAPI ou perf) est difficile. Il est nécessaire de connaître l'adresse des compteurs, de savoir comment les configurer (numéro de l'évènement, masque) et enfin d'interpréter les résultats. Ce travail est conséquent, car ces informations se trouvent dans de longues documentations (celle d'Intel fait plusieurs milliers de pages). Aussi, les descriptions de ces événements ne sont pas tout le temps clair, et trouver le bon évènement est complexe. Dans la même documentation, le même compteur peut être référencé avec le préfixe `IA32_` ou `MSR_` et certains registres `MSR` peuvent changer de nom d'une architecture à l'autre, rendant la recherche d'autant plus difficile. Certains détails des PMU Skylake (version 3) sont à aller chercher dans la version 1 plusieurs dizaines de pages avant, pour ensuite comprendre qu'elles ont été modifiées dans la version 2. De plus la documentation est incomplète, certains compteurs n'y apparaissent pas. Par exemple les compteurs *uncore* utilisés pour mesurer la bande passante mémoire ont été rendus publics plusieurs mois après dans une documentation séparée.

Face à ce manque de documentation il est légitime de se demander pourquoi celle-ci n'est pas réalisée avec plus de consistances. Un premier argument peut être de protéger la confidentialité de certaines implémentations du processeur. Une documentation fine des compteurs et du fonctionnement du processeur pourrait dévoiler certaines spécificités techniques, pouvant aider les autres fabricants. Une autre explication peut aussi venir de la vente de licences des outils propriétaires des constructeurs qui s'assurent une faible concurrence. Cependant, après plusieurs échanges avec les architectes de processeurs Intel, il semble que la principale raison soit technologique. Ajouter des compteurs implique l'utilisation de transistors supplémentaires (et donc d'énergie) pour une utilisation autre que l'amélioration de la performance du processeur. A l'inverse, l'ajout de compteurs aura un impact sur la performance des applications, notamment aux endroits sensibles tels que les caches. Les constructeurs semblent réticents à l'idée de substituer ces transistors qui peuvent être utilisés pour le développement d'autres matériels du processeur. Il est important de rappeler que ces compteurs ne sont à l'origine créés que pour des usages de débogage. Les constructeurs construisent alors le sous-ensemble minimum de compteurs leur permettant de vérifier le bon fonctionnement de la microarchitecture.

Validation des compteurs Pour pouvoir tirer des conclusions des résultats mesurés par les compteurs il est nécessaire de réaliser deux validations : la première est de s'assurer que l'évènement utilisé mesure bien le comportement souhaité, la deuxième est de valider le bon comportement du compteur.

Les valeurs reportées par les compteurs peuvent fortement varier entre deux exécutions de la même application. Même si un programme est spécifiquement conçu pour être déterministe, d'autres facteurs peuvent causer différents comptages (espace d'adressage différent, autre application utilisant une ressource partagée comme le cache). Le biais est une grande difficulté à appréhender lorsque les mesures peuvent varier d'un facteur trois pour la même application.

La vérification des évènements est quasi inexistante aujourd'hui. Ces efforts sont souvent réalisés en interne et la communauté [MVJ11] peut regretter l'inexistence de projet libre mettant en commun différents microbenchmarks permettant de valider précisément le bon fonctionnement des compteurs pour un ensemble d'évènements. [MVJ11] avance une idée pour les valider qui serait d'écrire un microbenchmark par évènement pour le stresser et avoir une fourchette des valeurs atteignables. Plusieurs travaux sont présentés dans leur papier pour valider les compteurs de miss du cache L1 en utilisant plusieurs compteurs censés les mesurer. Ils ne sont pas parvenus à atteindre les résultats attendus.

Interprétation des résultats La validation du comportement des compteurs est une première étape, mais il faut ensuite pouvoir en comprendre la signification. À partir de quel seuil le nombre d'évènements mesuré est-il problématique ? Les informations sous forme de ratio comme « nombre de miss par cycle » sont très dures à interpréter et nous pouvons nous demander à partir de quelle valeur elle peut être considérée comme bonne ou mauvaise. Ce genre de raisonnement peut être appliqué à la majorité des évènements mesurables, car la documentation des processeurs manque de préciser quelle valeur constitue le ratio d'une application ayant une bonne/mauvaise performance. Une idée pourrait être de définir des compteurs et leur associer des bonnes ou mauvaises valeurs, mais [MVJ11] fait remarquer que c'est un défi difficile à réaliser. Intel PMT [Rob11] utilise des techniques d'apprentissage pour corréler la valeur des compteurs avec les instructions exécutées.

Complexité de programmation. Sans recours à des interfaces hauts niveaux (papi, perf), la programmation des compteurs matériels est très complexe. Il faut développer du code bas niveau, assembleur/c qui varie d'un modèle de processeurs à l'autre rendant difficile la portabilité des outils développés. Le code bas niveau nécessaire pour leur programmation nécessite une lecture attentive de la documentation du constructeur qui fait plusieurs milliers de pages et qui n'est pas toujours complète. Des architectures récentes comme les processeurs Intel Skylake possèdent des dizaines de compteurs permettant de suivre l'activité des contrôleurs mémoires. La sélection du compteur approprié pour résoudre un problème en particulier est loin d'être trivial, notamment dû au manque de documentation de ceux-ci.

Portabilité des outils. La majorité des évènements des architectures Intel sont considérés comme non architecturaux. Intel ne garantit pas que ces compteurs soient maintenus dans les prochaines versions de processeur. Cette particularité est une raison majeure du renoncement des développeurs à écrire et/ou à maintenir des outils qui pourraient ne plus être utilisables sur l'architecture suivante. Pour notre travail, une difficulté est de trouver des outils portables sur des architectures différentes dont la communauté soit suffisamment active pour porter leur développement. Certains outils fonctionnels ont été développés il y a plusieurs années, mais l'effort de portage n'a jamais été réalisé. Ce critère est essentiel dans le choix des outils utilisés dans notre travail qui a pour ambition de pouvoir être appliqué sur des architectures de différents constructeurs.

Droits La configuration des compteurs n'est possible qu'en utilisant les instructions `rdmsr` et `wrmsr`. Pour des raisons de sécurité, il semble évident que de laisser l'accès à ces informations aux utilisateurs est dangereux, notamment lorsque les processeurs sont partagés entre plusieurs utilisateurs. Cependant, l'obligation d'avoir des droits privilégiés pour utiliser les compteurs est un frein au développement d'outils, mais aussi à l'utilisation de ces compteurs. En effet, dans des environnements tels que les centres de calculs, il est rare que les utilisateurs possèdent ce niveau de droits. Bien que ce soit rare dans le domaine du HPC, les centres de données ont de plus en plus recours à la virtualisation ou à la dockerisation. L'ajout de cette couche rend d'autant plus complexes l'accès et la valorisation de ces compteurs.

Impact sur la performance L'incrémentation des compteurs de performance n'affecte pas (ou peu [Röh+15]) la performance de l'application étudiée. L'impact sur la performance vient de l'extraction de ces valeurs pour pouvoir être traité. Comme dans tout système physique, l'acte de mesurer perturbe le phénomène mesuré. Les accès aux compteurs à travers des interfaces introduisent nécessairement des coûts sous la forme d'instructions supplémentaires (y compris les appels système), et les interfaces provoquent une pollution du cache qui peut modifier le comportement du cache et de la mémoire de l'application surveillée [Wea15]. Certains travaux ont été menés pour mesurer le coût de l'utilisation des compteurs matériel [BN14]. Les résultats montrent que le coût varie fortement en fonction des événements étudiés, de la fréquence d'accès aux compteurs, du nombre d'événements et surtout de l'utilisation du mode d'échantillonnage. L'impact le plus fort sur la performance a été mesuré lors de l'utilisation du multiplexage. Il est notamment souvent recommandé de ne pas utiliser plus d'événements qu'il n'y a de compteurs matériels disponibles, en particulier en mode échantillonnage. Sinon, le coût engendré peut atteindre jusqu'à 25% d'augmentation du temps d'exécution de l'application. Ce résultat suggère qu'il y a encore de la place pour l'optimisation des interfaces d'accès aux compteurs.

B.4 Conclusion

Les compteurs matériels de performance sont un outil puissant qui permet d'avoir un aperçu du comportement des microarchitectures qui se sont grandement complexifiées ces dernières

années. Malheureusement l'adoption à grande échelle des compteurs matériels a été ralentie par le manque de documentation et le manque d'interfaces multi-plateformes. Historiquement, les compteurs matériels étaient utilisés en interne par les concepteurs de puces et les interfaces n'étaient pas toujours documentées ou mises à la disposition du public. Pour ces raisons, nous estimons qu'il est important de présenter les compteurs et les moyens les plus efficaces de les utiliser. Aujourd'hui encore, ce domaine souffre d'un manque de documentation et d'exemples. D'autres freins ont rendu le développement des outils de profilage difficile comme le manque d'interface unique pour les différentes plateformes ou la nécessité d'avoir des droits privilégiés (noyau) pour accéder aux compteurs.

Ce travail de thèse s'intéresse aux compteurs matériels pour deux raisons principales : l'analyse des performances et l'optimisation des applications. L'objectif de cette section est de réaliser un état de l'art des outils existant pour notre démarche. L'utilisation des compteurs matériels a de nombreux avantages, mais aussi des limites qui empêchent l'automatisation de l'analyse de performance.

Bibliographie

[Abb+09] Benjamin P Abbott et al. « Einstein@ Home search for periodic gravitational waves in early S5 LIGO data ». In : *Physical review d* 80.4 (2009), p. 42003 (cf. p. 20).

[ABB64] Gene M Amdahl, Gerrit A Blaauw et F P Brooks. « Architecture of the IBM System/360 ». In : *IBM Journal of Research and Development* 8.2 (1964), p. 87–101 (cf. p. 234).

[Adr19] Adrian BRANCO. *Samsung prépare l'arrivée des puces gravées en 3 nm pour 2021*. 2019. URL : Liens (cf. p. 232).

[AH18] Dario Amodei et Danny Hernandez. *AI and Compute, May 2018*. 2018 (cf. p. 46).

[Ahn+09] Jung Ho Ahn et al. « HyperX : topology, routing, and packaging of efficient large-scale networks ». In : *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, p. 1–11 (cf. p. 83).

[AlH+10] Samaher Al-Hothali et al. « Snoopy and Directory Based Cache Coherence Protocols : A Critical Analysis ». In : *Journal of Information & Communication Technology* 4.1 (2010), p. 1–10. URL : Liens (cf. p. 271).

[Alv10] Jérémy Alvarez-Hérault. *Mémoire magnétique à écriture par courant polarisé en spin assistée thermiquement*. 2010 (cf. p. 68).

[Ama+15] Jose N Amaral et al. *Identification of performance bottlenecks*. Août 2015 (cf. p. 173).

[Amd67] Gene M Amdahl. « Validity of the single processor approach to achieving large scale computing capabilities ». In : *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, p. 483–485 (cf. p. 32).

[AMM04] Hussein Al-Zoubi, Aleksandar Milenkovic et Milena Milenkovic. « Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite ». In : *Proceedings of the 42Nd Annual Southeast Regional Conference*. ACM-SE 42. New York, NY, USA : ACM, 2004, p. 267. URL : Liens (cf. p. 269).

[And+19] Hugo Andrade et al. « Software Challenges in Heterogeneous Computing : A Multiple Case Study in Industry ». In : 2019 (cf. p. 77).

[Ang16] James A Ang. *Exascale System and Node Architectures*. Rapp. tech. 2016 (cf. p. 49, 50).

[Ant18] Anthony Nelzin-Santos. *10 nm, 7 nm, 5 nm : la finesse de gravure, enjeu du monde mobile*. 2018. URL : Liens (cf. p. 231).

[Asa+06] Krste ASANOVIC et al. *The landscape of parallel computing research : A view from berkeley*. Rapp. tech. Technical Report UCB/EECS-2006-183, EECS Department, University of˜..., 2006 (cf. p. 74).

[asp] ASPSYS. *Intel Xeon Scalable Family of Processors - High Performance Computers | Aspen Systems*. URL : Liens (cf. p. 194).

[Bai+14] Brian BAILLARGEON et al. « The living heart project : A robust and integrative simulator for human heart function ». In : *European Journal of Mechanics, A/Solids* 48.1 (2014), p. 38–47 (cf. p. 1).

[Bar+10] Denis BARTHOU et al. « Performance tuning of x86 openmp codes with maqao ». In : *Tools for High Performance Computing 2009*. Springer, 2010, p. 95–113 (cf. p. 173).

[Bar+12] R. F. BARRETT et al. « Navigating an evolutionary fast path to exascale ». In : *Proceedings - 2012 SC Companion : High Performance Computing, Networking Storage and Analysis, SCC 2012*. IEEE. 2012, p. 355–365 (cf. p. 56).

[Bar82] D. F. BARBE. *Very Large Scale Integration (VLSI) Fundamentals and Applications*. T. 4. Springer Science & Business Media, 1982 (cf. p. 228).

[BB48] John BARDEEN et Walter Hauser BRATTAIN. « The transistor, a semi-conductor triode ». In : *Physical Review* 74.2 (1948), p. 230 (cf. p. 226).

[BC18] Avishek BISWAS et Anantha P CHANDRAKASAN. « Conv-RAM : An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications ». In : *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018, p. 488–490 (cf. p. 62).

[BC91] Jean-Loup BAER et Tien-Fu CHEN. « An effective on-chip preloading scheme to reduce data access penalty ». In : *Supercomputing'91 : Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE. 1991, p. 176–186 (cf. p. 257).

[BDI13] Gérard BLANCHET, Bertrand DUPOUY et IMPR. SEPEC). *Architecture des ordinateurs : principes fondamentaux*. Hermes Science publications-Lavoisier, 2013 (cf. p. 267, 270).

[Bel+17] N. L. BELYAEV et al. *High performance computing system in the framework of the Higgs boson studies at ATLAS*. Rapp. tech. 2017, p. 23–29 (cf. p. 1).

[Ben+14a] Siegfried BENKNER et al. « Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401) ». In : *Dagstuhl Reports* 3.9 (2014). Sous la dir. de Siegfried BENKNER et al., p. 214–244. URL : Liens (cf. p. 77).

[Ben+14b] Ed BENNETT et al. « BSMBench : a flexible and scalable supercomputer benchmark from computational particle physics ». In : (2014). URL : Liens (cf. p. 88).

[Ber+11] Keren Bergman et al. « Let there be light ! : the future of memory systems is photonics and 3D stacking ». In : *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. ACM, 2011, p. 43–48 (cf. p. 47).

[Ber+13] Peter Berczik et al. « Up to 700k GPU Cores, Kepler, and the Exascale Future for Simulations of Star Clusters Around Black Holes ». In : *Supercomputing*. Sous la dir. de Julian Martin Kunkel, Thomas Ludwig et Hans Werner Meuer. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 13–25 (cf. p. 44).

[Ber15] Keren Bergman. *PIC training : Interconnect System Design*. Rapp. tech. 2015 (cf. p. 69, 70).

[Ber18] Keren Bergman. « Empowering Flexible and Scalable High Performance Architectures with Embedded Photonics ». In : *IPDPS 2018*. IPDPS 2018, 2018, p. 378–378 (cf. p. 43, 47, 71, 79).

[Bet+03] A Bette et al. « A high-speed 128 Kbit MRAM core for future universal memory applications ». In : *VLSI Circuits, 2003. Digest of Technical Papers. 2003 Symposium on*. IEEE, 2003, p. 217–220 (cf. p. 68).

[BGL00] Amol Bakshi, JL Gaudiot et WY Lin. « Memory Latency : to tolerate or to reduce ». In : *12th Symposium on ...* (2000). URL : Liens (cf. p. 257).

[BLK18] James Bucek, Klaus-Dieter Lange et Jóakim v. Kistowski. « SPEC CPU2017 : Next-generation compute benchmark ». In : *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, p. 41–42 (cf. p. 7, 90).

[Blo+06] David L Block et al. « An almost head-on collision as the origin of two off-centre rings in the Andromeda galaxy ». In : *Nature* 443.7113 (2006), p. 832. URL : Liens (cf. p. 18).

[Blo+17] Jakob Blomer et al. « Delivering LHC Software to HPC Compute Elements with CernVM-FS ». In : *High Performance Computing*. Sous la dir. de Julian M Kunkel et al. Cham : Springer International Publishing, 2017, p. 724–730 (cf. p. 43).

[BMP16] Alexandre Bonhomme, Philippe Mathieu et Sébastien Picault. « Simuler le trafic routier à partir de données réelles ». In : *Revue des Sciences et Technologies de l'Information - Série RIA : Revue d'Intelligence Artificielle* 30.3 (juin 2016), p. 329–352. URL : Liens (cf. p. 44).

[BN14] Georgios Bitzes et Andrzej Nowak. « The overhead of profiling using PMU hardware counters ». In : (2014) (cf. p. 299).

[Bra82] A. Brandt. « Introductory remarks on multigrid methods. » In : (1982) (cf. p. 57).

[Bri] Britannica. *IBM OS/360 | operating system | Britannica.com*. URL : Liens (cf. p. 273).

[Bro+00] S. Browne et al. « A portable programming interface for performance evaluation on modern processors ». In : *International Journal of High Performance Computing Applications* 14.3 (août 2000), p. 189–204. URL : Liens (cf. p. 9, 103, 168, 169).

[Bro17] Rodney Brooks. *The End of Moore's Law – Rodney Brooks*. 2017. URL : Liens (cf. p. 39).

[BW13] Holger Brunst et Matthias Weber. « Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite ». In : *Tools for High Performance Computing 2012*. Springer, 2013, p. 95–114 (cf. p. 199, 210).

[CA04] Yuan Chou et Santosh Abraham. *Efficient register file checkpointing to facilitate speculative execution*. 2004 (cf. p. 260).

[Cas+15] Pablo De Oliveira Castro et al. « CERE : LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization ». In : *ACM Transactions on Architecture and Code Optimization* 12.1 (2015), p. 1–24. URL : Liens (cf. p. 77, 199).

[Cav+06] John Cavazos et al. « Automatic performance model construction for the fast software exploration of new hardware designs ». In : *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, p. 24–34 (cf. p. 97).

[CEA07] CEA. *La simulation numérique*. Rapp. tech. 2007. URL : Liens (cf. p. 17).

[Cha+14] Andres S Charif-Rubial et al. « CQA : A Code Quality Analyzer tool at binary level using performance modeling and evaluation ». In : *21th Annual International Conference on High Performance Computing - HiPC'14*. IEEE. 2014, p. 1–10. URL : Liens (cf. p. 96, 173).

[Che+16] Fei Chen et al. *Billion node graph inference : iterative processing on The Machine*. Rapp. tech. 2016 (cf. p. 85).

[Chr17] Chris Evans. *Has NVMe Killed off NVDIMM ? | Architecting IT*. 2017. URL : Liens (cf. p. 66).

[Chu+12] I-Hsin Chung et al. *Automated detection of application performance bottlenecks*. Juil. 2012 (cf. p. 116).

[Chu71] Leon Chua. « Memristor-the missing circuit element ». In : *IEEE Transactions on circuit theory* 18.5 (1971), p. 507–519 (cf. p. 62).

[CK95] Bob Cmelik et David Keppel. « Shade : A fast instruction-set simulator for execution profiling ». In : *Fast simulation of computer architectures*. Springer, 1995, p. 5–46 (cf. p. 94).

[CM90] John Cocke et Victoria Markstein. « The evolution of RISC technology at IBM ». In : *IBM Journal of research and development* 34.1 (1990), p. 4–11 (cf. p. 237).

[CML14] Min Chen, Shiwen Mao et Yunhao Liu. « Big Data : A Survey ». In : *Mobile Networks and Applications* 19.2 (avr. 2014), p. 171–209. URL : Liens (cf. p. 45).

[Coo97] Intel Coorporation. « Using the rdtsc instruction for performance monitoring ». In : *Techn. Ber., tech. rep., Intel Coorporation* (1997), p. 22 (cf. p. 154).

[Cor16] James W. Cortada. *The Computer in the United States*. Routledge, 2016 (cf. p. 234).

[Cou09] HPC Advisory Council. « Interconnect Analysis : 10GigE and InfiniBand in High Performance Computing ». In : *Case Studies* (2009) (cf. p. 25).

[CX14] Jason Cong et Bingjun Xiao. « Minimizing computation in convolutional neural networks ». In : *International conference on artificial neural networks*. Springer, 2014, p. 281–290 (cf. p. 57).

[D+97] Jack J Dongarra, Hans W Meuer, Erich Strohmaier et al. « TOP500 supercomputer sites ». In : *Supercomputer* 13 (1997), p. 89–111 (cf. p. 89).

[Dai+05] Xiaoru Dai et al. « A general compiler framework for speculative optimizations using data speculative code motion ». In : *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, p. 280–290 (cf. p. 97).

[Dat+08] Kaushik Datta et al. « Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures ». In : *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008*. IEEE Press. 2008, p. 4 (cf. p. 77, 111).

[DB05] Lamia Djoudi et Denis Barthou. « Maqao : Modular assembler quality analyzer and optimizer for itanium 2 ». In : *Workshop on EPIC architectures and compiler technology*. 2005. URL : Liens (cf. p. 96, 168).

[De 10] Arnaldo Carvalho De Melo. « The new linux'perf'tools ». In : *Slides from Linux Kongress*. T. 18. 2010 (cf. p. 96, 199).

[Den+18] Nicolas Denoyelle et al. « Modeling large compute nodes with heterogeneous memories with cache-aware roofline model ». In : *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. T. 10724 LNCS. 2018, p. 91–113 (cf. p. 109).

[Den+74] R. H. Dennard et al. « Design for ion-implanted MOSFET's with very small physical dimensions ». In : *IEEE Journal of Solid-State Circuits* 9.5 (oct. 1974), p. 256–268. URL : Liens (cf. p. 3, 39, 242).

[DH13] Jack Dongarra et Ma Heroux. « Toward a new metric for ranking high performance computing systems ». In : *Sandia Report, SAND2013-4744* June (2013), p. 1–19. URL : Liens (cf. p. 7, 8, 89).

[Dil12] Patel Hardik Dilipbhai. *Understanding the Challenges in Parallelization of Selected SPEC CINT2006 Applications*. 2012 (cf. p. 90).

[DLP03] Jack J. Dongarra, Piotr Luszczek et Antoine Petite. « The LINPACK benchmark : Past, present and future ». In : *Concurrency Computation Practice and Experience* 15.9 (août 2003), p. 803–820. URL : Liens (cf. p. 2, 4, 8, 34, 41, 45, 89, 114, 145, 196).

[Dol15] Romain Dolbeau. « Theoretical Peak FLOPS per instruction set on less conventional hardware ». In : (2015). URL : Liens (cf. p. 192).

[Dre07] U Drepper. « What every programmer should know about memory ». In : *Red Hat* (2007), p. 1–114. URL : Liens (cf. p. 166, 268).

[Dro07] Paul J Drongowski. « Instruction-based sampling : A new performance analysis technique for AMD family 10h processors ». In : (2007) (cf. p. 97, 285).

[DS08] Alexandre X Duchateau et Albert Sidelnik. « P-Ray : A Suite of Micro-benchmarks for Multi-core Architectures ». In : (2008) (cf. p. 93).

[DZ83] John D Day et Hubert Zimmermann. « The OSI reference model ». In : *Proceedings of the IEEE* 71.12 (1983), p. 1334–1340 (cf. p. 225).

[ECT17] Danijela Efnusheva, Ana Cholakoska et Aristotel Tentov. « A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck ». In : *International Journal of Computer Science and Information Technology*. T. 9. 2. 2017, p. 151–163 (cf. p. 42, 256, 257).

[ED87] Philip G. Emma et Edward S. Davidson. « Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance ». In : *IEEE Transactions on Computers* C-36.7 (1987), p. 859–875 (cf. p. 249).

[Edg30] Lilienfeld Julius Edgar. *Method and apparatus for controlling electric currents*. 1930 (cf. p. 226).

[Eij13] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2013 (cf. p. 28).

[Era06] Stéphane Eranian. « Perfmon2 : a flexible performance monitoring interface for Linux ». In : *Proc of the 2006 Ottawa Linux Symposium* (2006), p. 269–288. URL : Liens (cf. p. 99).

[Eur18] EuroHPC. *Commission Européenne - Communiqué de presse HPC et EuroHPC*. 2018. URL : Liens (cf. p. 59).

[EV97] Roger Espasa et Mateo Valero. « Exploiting instruction-and data-level parallelism ». In : *IEEE micro* 17.5 (1997), p. 20–27 (cf. p. 28).

[Far14] Asma Farjallah. « Preparing depth imaging applications for Exascale challenges and impacts ». Thèse de doct. Université de Versailles-Saint Quentin en Yvelines, 2014 (cf. p. 109).

[FG13] Alexander Frolov et Mikhail Gilmendinov. « DISBench : Benchmark for Memory Performance Evaluation of Multicore Multiprocessors ». In : t. 7979. 2013, p. 197–207 (cf. p. 120).

[FJ98] Matteo Frigo et Steven G. Johnson. « FFTW : An adaptive software architecture for the FFT ». In : *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. T. 3. IEEE. 1998, p. 1381–1384 (cf. p. 88).

[Fly11] Michael Flynn. « Flynn's taxonomy ». In : *Encyclopedia of parallel computing* (2011), p. 689–697 (cf. p. 27).

[Fra15] François Bodin. *La fin de la Loi de Moore et les applications HPC*. Rapp. tech. Rennes : LIP6, 2015. URL : Liens (cf. p. 39).

[FW08] R. F. Freitas et W. W. Wilcke. « Storage-class memory : The next storage system technology ». In : *IBM Journal of Research and Development* 52.4.5 (juil. 2008), p. 439–447. URL : Liens (cf. p. 64).

[Ger18] Gerald Huguenin. *Transistor bipolaire*. 2018. URL : Liens (cf. p. 226).

[Ger19] Bill Gervasi. « Will Carbon Nanotube Memory Replace DRAM ? » In : *IEEE Micro* 39.2 (2019), p. 45–51 (cf. p. 69).

[Gon+10] Jorge González-Dominguez et al. « Servet : A benchmark suite for autotuning on multicore clusters ». In : *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*. IEEE. 2010, p. 1–9 (cf. p. 7, 93).

[GR15] Jha Shantenu Geoffrey Fox et Lavanya Ramakrishnan. « First workshop on streaming and steering applications : Requirements and infrastructure ». In : (2015), p. 1–24. URL : Liens (cf. p. 45).

[GTW19] Fei Gao, Georgios Tziantzioulis et David Wentzlaff. « ComputeDRAM : In-memory compute using off-the-shelf DRAMs ». In : *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, p. 100–113 (cf. p. 62).

[Gun20] Thomas Guntz. « Estimating Expertise from Eye Gaze and Emotions ». Thèse de doct. 2020 (cf. p. v).

[Gus88] John L Gustafson. « Reevaluating Amdahl's law ». In : *Communications of the ACM* 31.5 (1988), p. 532–533 (cf. p. 34).

[GY17] Kartikay Garg et Jeffrey Young. « Evaluating hybrid memory cube infrastructure to support high-performance sparse algorithms ». In : *Proceedings of the International Symposium on Memory Systems*. ACM, 2017, p. 112–114 (cf. p. 55).

[Han15] Jim Handy. « Understanding the Intel/Micron 3D XPoint memory ». In : *Proc. SDC* (2015) (cf. p. 68).

[HE08] Kenneth Hoste et Lieven Eeckhout. « COLE : Compiler Optimization Level Exploration Categories and Subject Descriptors ». In : *ACM conference on code generation and optimization*. ACM. 2008, p. 165–174 (cf. p. 77).

[HG71] D J Hatfield et J Gerald. « Program Restructuring for Virtual Memory ». In : *IBM Syst. J.* 10.3 (sept. 1971), p. 168–192. URL : Liens (cf. p. 262).

[Hir12] I Hirsh. *Intel architecture code analyzer*. 2012 (cf. p. 96).

[Hor14] Mark Horowitz. « 1.1 computing's energy problem (and what we can do about it) ». In : *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, p. 10–14 (cf. p. 50, 74).

[HPE16] HPE. « Exascale : A race to the future of HPC ». In : *HP white paper* (2016) (cf. p. 3).

[Hur14] Patrick J Hurley. *A concise introduction to logic*. Nelson Education, 2014 (cf. p. 227).

[IBM13] IBM. « Storage Class Memory : Towards a disruptively low-cost solid-state non-volatile memory ». In : *IBM Almaden Research Center* (2013). URL : Liens (cf. p. 65).

[ICM05] Canturk Isci, Gilberto Contreras et Margaret Martonosi. « Hardware performance counters for detailed runtime power and thermal estimations : Experiences and proposals ». In : *Proceedings of the Hardware Performance Monitor Design and Functionality Workshop in the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*. 2005 (cf. p. 97).

[Ima+11] Khadija Imadoueddine et al. « Tips and tricks for finite difference and i/o less FWI ». In : *SEG Technical Program Expanded Abstracts 2011*. Society of Exploration Geophysicists, 2011, p. 3174–3178 (cf. p. 94).

[Int11] Intel. « Intel 64 and ia-32 architectures software developer's manual ». In : *Volume 3b : System Programming Guide (Part 2)* 2 (2011), p. 14–19 (cf. p. 275, 276).

[Int17] Intel. *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual*. 2017 (cf. p. 294, 295).

[Int18] Intel. « Intel 64 and IA-32 architectures software developer's manual combined volumes : 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 ». In : (2018) (cf. p. 233, 284, 288).

[IPS14] Aleksandar Ilic, Frederico Pratas et Leonel Sousa. « Cache-aware roofline model : Upgrading the loft ». In : *IEEE Computer Architecture Letters* 13.1 (2014), p. 21–24 (cf. p. 109).

[JK12] Joe Jeddeloh et Brent Keeth. « Hybrid memory cube new DRAM architecture increases density and performance ». In : *2012 symposium on VLSI technology (VLSIT)*. IEEE, 2012, p. 87–88 (cf. p. 63).

[Joh10] John McCalpin. *John McCalpin's blog » Blog Archive » Optimizing AMD Opteron Memory Bandwidth, Part 1 : single-thread, read-only*. 2010. URL : Liens (cf. p. 197).

[Joh17] John LeDuc. *Principes de base des transistors*. 2017. URL : Liens (cf. p. 226).

[Joh89] William M Johnson. *Super-scalar processor design*. 1989 (cf. p. 250).

[Jou+17] Norman P Jouppi et al. « In-datacenter performance analysis of a tensor processing unit ». In : *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, p. 1–12 (cf. p. 74).

[Kam+09] Prabhanjan Kambadur et al. « PFunc : modern task parallelism for modern high performance computing ». In : *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, p. 1–11 (cf. p. 28).

[Kan+18] Mingu Kang et al. « A multi-functional in-memory inference processor using a standard 6T SRAM array ». In : *IEEE Journal of Solid-State Circuits* 53.2 (2018), p. 642–655 (cf. p. 62).

[Kha13] Amal Khabou. « Dense matrix computations : communication cost and numerical stability ». Thèse de doct. 2013 (cf. p. 58).

[Kha19] Hamidreza Khaleghzadeh. « Novel Data-Partitioning Algorithms for Performance and Energy Optimization of Data-Parallel Applications on Modern Heterogeneous HPC Platforms ». Thèse de doct. 2019 (cf. p. 77).

[Knu71] Donald E Knuth. « An empirical study of FORTRAN programs ». In : *Software : Practice and experience* 1.2 (1971), p. 105–133 (cf. p. 95).

[Koc+18] Paul Kocher et al. « Spectre Attacks : Exploiting Speculative Execution ». In : *arXiv preprint arXiv :1801.01203* (2018). URL : Liens (cf. p. 53, 246).

[Kog+08] Peter Kogge et al. « ExaScale Computing Study : Technology Challenges in Achieving Exascale Systems ». In : *(P. Kogge, Editor and Study Lead)* TR-2008-13 (2008), p. 1–278. URL : Liens (cf. p. 3, 47, 70).

[Kot16] Douglas B Kothe. « Exascale Applications : Opportunities and Challenges ». In : (2016) (cf. p. 114).

[Kro+14] Dirk P Kroese et al. « Why the Monte Carlo Method is so important today Uses of the MCM ». In : *WIREs Computational Statistics* 6.6 (2014), p. 386–392. URL : Liens (cf. p. 16).

[Kur01] Tadahiro Kuroda. « CMOS design challenges to power wall ». In : *Digest of Papers. Microprocesses and Nanotechnology 2001. 2001 International Microprocesses and Nanotechnology Conference (IEEE Cat. No. 01EX468)*. IEEE, 2001, p. 6–7 (cf. p. 40).

[Lar+09] Stefan M Larson et al. « Folding@ Home and Genome@ Home : Using distributed computing to tackle previously intractable problems in computational biology ». In : *arXiv preprint arXiv :0901.0866* (2009) (cf. p. 21).

[Lar16] Florian Larysch. « Fine-grained estimation of memory bandwidth utilization ». Thèse de doct. 2016 (cf. p. 167).

[Lat16] Chris Lattner. *Clang Static Analyzer*. 2016. URL : Liens (cf. p. 96).

[LE04] John Levon et Philippe Elie. *Oprofile : A system profiler for linux*. 2004 (cf. p. 103).

[LeC89] Yann LeCun. « Generalization and network design strategies ». In : *Connectionism in perspective* (1989), p. 143–155 (cf. p. 43).

[Lel+14] Robert Leland et al. « Large-Scale Data Analytics and Its Relationship to Simulation ». In : (2014) (cf. p. 50, 51).

[LG08] John D C Little et Stephen C Graves. « Little's law ». In : *Building intuition*. Springer, 2008, p. 81–100 (cf. p. 128, 194, 197).

[LGC97] Jesus Labarta, Sergi Girona et Toni Cortes. « Analyzing scheduling policies using Dimemas ». In : *Parallel Computing* 23.1-2 (1997), p. 23–34 (cf. p. 9, 105).

[Li+16] Hao Li et al. « Pruning filters for efficient convnets ». In : *arXiv preprint arXiv :1608.08710* (2016) (cf. p. 57).

[Li+17] Shuangchen Li et al. « Drisa : A dram-based reconfigurable in-situ accelerator ». In : *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, p. 288–301 (cf. p. 62).

[Lim+] Robert Lim et al. « Computationally efficient multiplexing of events on hardware counters ». In : *Linux Symposium*. Citeseer, p. 101–110 (cf. p. 290).

[Lin] Linux. *gettimeofday(2) - Linux manual page*. URL : Liens (cf. p. 206).

[Lin+00] Kathleen A Lindlan et al. « A tool framework for static and dynamic analysis of object-oriented software with templates ». In : *SC'00 : Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000, p. 49 (cf. p. 168).

[Lin19a] Linux. *Linux Kernel Huge Pages*. 2019. URL : Liens (cf. p. 277).

[Lin19b] Linux. *Linux Kernel Huge Pages Test*. 2019. URL : Liens (cf. p. 277).

[Lin19c] Linux. « Transparent Hugepage Support ». In : *GitHub repository*. 2019. URL : Liens (cf. p. 277).

[Lip+12] Benjamin Lipshitz et al. « Communication-avoiding parallel strassen : Implementation and performance ». In : *SC'12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, p. 1–11 (cf. p. 58).

[Lip+18] Moritz Lipp et al. « Meltdown ». In : *CoRR* abs/1801.0 (2018). URL : Liens (cf. p. 245).

[Lit61] John D C Little. « A Proof for the Queuing Formula : L = A W ». In : *Operations Research* 9.3 (juin 1961), p. 383–387. URL : Liens (cf. p. 263).

[Liu16] Hao Liu. « Protocoles scalables de cohérence des caches pour processeurs manycore à espace d'adressage partagé visant la basse consommation. » Thèse de doct. Paris 6, 2016 (cf. p. 271).

[Lo+15] Yu Jung Lo et al. « Roofline model toolkit : A practical tool for architectural and program analysis ». In : *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. T. 8966. Springer. 2015, p. 129–148 (cf. p. 109).

[Luc+14] Robert Lucas et al. « Top ten exascale research challenges ». In : *DOE ASCAC subcommittee report* (2014), p. 1–86 (cf. p. 3, 47, 58, 72).

[Lus+06] Piotr R Luszczek et al. « The HPC Challenge (HPCC) benchmark suite ». In : *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. T. 213. Citeseer, 2006 (cf. p. 7, 8, 90).

[Mar+02] Deborah T Marr et al. « Hyper-Threading Technology Architecture and Microarchitecture ». In : *Intel Technology Journal* 6.1 (2002), p. 1–12. URL : Liens (cf. p. 162).

[Mar12] Henry Markram. « The human brain project ». In : *Scientific American* 306.6 (2012), p. 50–55 (cf. p. 1).

[MAR14] Christian Märtin, Hochschule Augsburg et Fachgebiete Rechnerarchitektur. « Post-Dennard Scaling and the final Years of Moore's Law Consequences for the Evolution of Multicore-Architectures ». In : *Informatik und Interaktive Systeme* (2014) (cf. p. 241, 242).

[Mat14] Pär Persson Mattsson. *Why Haven't CPU Clock Speeds Increased in the Last Few Years ?* 2014. URL : Liens (cf. p. 251).

[McC95] John D McCalpin. « STREAM benchmark ». In : *Link : www. cs. virginia. edu/stream/ref. html# what* 22 (1995) (cf. p. 7, 88, 91, 114, 141, 166, 170, 190, 196).

[Met18] Cade Metz. *Big Bets on A.I. Open a New Frontier for Chip Start-Ups, Too*. 2018. URL : Liens (cf. p. 74).

[Meu17] Quentin Meunier. *Introduction Fonctionnement des mémoires cache Optimisation des mémoires cache Fonctionnement et Optimisation des Mémoires Cache*. Rapp. tech. 2017. URL : Liens (cf. p. 267, 268).

[MG91] Todd Mowry et Anoop Gupta. « Tolerating latency through software-controlled prefetching in shared-memory multiprocessors ». In : *Journal of parallel and Distributed Computing* 12.2 (1991), p. 87–106 (cf. p. 257).

[MMK02] Milena MILENKOVIC, Aleksandar MILENKOVIC et Jeffrey KULICK. « Demystifying Intel branch predictors ». In : *Proceedings of the 2002 Workshop on Duplicating, Deconstructing and Debunking (WDDD'02)* Figure 1 (2002) (cf. p. 246).

[Mol+17] Daniel MOLKA et al. « Detecting memory-boundedness with hardware performance counters ». In : *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. New York, New York, USA : ACM, 2017, p. 27–38. URL : Liens (cf. p. 92).

[Moo65] G E MOORE. « Cramming More Components Onto Integrated Circuits ». In : *Electronics* 38.8 (jan. 1965), p. 82–85. URL : Liens (cf. p. 39, 231).

[Moo75] Gordon E. MOORE. « Progress In Digital Integrated Electronics ». In : *IEEE Solid-State Circuits Newsletter* 20.3 (1975), p. 36–37. URL : Liens (cf. p. 3, 9, 39, 231).

[Mos06] Tipp MOSELEY. *Adaptive thread scheduling for simultaneous multithreading processors*. 2006 (cf. p. 97).

[MPV93] Mayan MOUDGILL, Keshav PINGALI et Stamatis VASSILIADIS. « Register renaming and dynamic speculation : an alternative approach ». In : *Proceedings of the 26th annual international symposium on Microarchitecture*. IEEE. 1993, p. 202–213 (cf. p. 260).

[MR12] Todd C MOWRY et Anthony ROWE. *Virtual Memory : Systems*. Rapp. tech. 2012. URL : Liens (cf. p. 279–281).

[MTB11] Abdelhafid MAZOUZ, Sid Ahmed Ali TOUATI et Denis BARTHOU. « Performance evaluation and analysis of thread pinning strategies on multi-core platforms : Case study of SPEC OMP applications on intel architectures ». In : *Proceedings of the 2011 International Conference on High Performance Computing and Simulation, HPCS 2011*. IEEE. 2011, p. 273–279 (cf. p. 77, 254).

[MVJ11] Tipp MOSELEY, Neil VACHHARAJANI et William JALBY. « Hardware performance monitoring for the rest of us : A position and survey ». In : *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6985 LNCS (2011), p. 293–312 (cf. p. 97, 98, 298).

[Nag+96] Wolfgang E NAGEL et al. « VAMPIR : Visualization and analysis of MPI resources ». In : (1996) (cf. p. 210).

[NCC17] Klara NAHRSTEDT, Christos G. CASSANDRAS et Charlie CATLETT. « City-Scale Intelligent Systems and Platforms ». In : (mai 2017). URL : Liens (cf. p. 45).

[Neu93] J von NEUMANN. « First draft of a report on the EDVAC ». In : *IEEE Annals of the History of Computing* 15.4 (1993), p. 27–75 (cf. p. 235).

[OCo+17] Mike O'CONNOR et al. « Fine-grained DRAM : energy-efficient DRAM for extreme bandwidth systems ». In : *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, p. 41–54 (cf. p. 64).

[Oli+05] Leonid Oliker et al. « A performance evaluation of the cray X1 for scientific applications ». In : *Lecture Notes in Computer Science* 3402 (2005), p. 51–65. URL : Liens (cf. p. 3, 41, 114).

[Ouk+15] Ismail Oukid et al. « Instant Recovery for Main Memory Databases. » In : *CIDR*. 2015 (cf. p. 64).

[Pal15] Vincent Palomares. « Combining static and dynamic approaches to model loop performance in HPC ». Thèse de doct. Université de Versailles-Saint Quentin en Yvelines, 2015 (cf. p. 94).

[Pan+18] Reena Panda et al. « Wait of a Decade : Did SPEC CPU 2017 Broaden the Performance Horizon ? » In : *Proceedings - International Symposium on High-Performance Computer Architecture*. T. 2018-Febru. 2018, p. 271–282 (cf. p. 91).

[PDC19] Jean Pourroy, Patrick Demichel et Denis Christophe. « Assembly micro-benchmark generator for characterizing Floating Point Units ». In : *HPCS 2019 - 17th International Conference on High Performance Computing & Simulation*. Dublin, Ireland : IEEE, juil. 2019 (cf. p. 12, 143).

[Pen+17] I B Peng et al. « Exploring the Performance Benefit of Hybrid Memory System on HPC Environments ». In : *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Mai 2017, p. 683–692 (cf. p. 64).

[Pet05] Mikael Pettersson. *Perfctr : the linux performance monitoring counters driver*. 2005 (cf. p. 99).

[Pil+95] Vincent Pillet et al. « Paraver : A tool to visualize and analyze parallel code ». In : *Proceedings of WoTUG-18 : transputer and occam developments*. T. 44. 1. 1995, p. 17–31 (cf. p. 9, 105, 189).

[PKV09] Li-Shiuan Peh, Stephen W. Keckler et Sriram Vangal. « On-Chip Networks for Multicore Systems ». In : *Multicore Processors and Systems*. Springer, 2009, p. 35–71 (cf. p. 254).

[Pop16] Mihail Popov. « Automatic decomposition of parallel programs for optimization and performance prediction. » Thèse de doct. Université Paris-Saclay, oct. 2016. URL : Liens (cf. p. 77, 96, 173).

[Pou+18] Jean Pourroy et al. « Calcul distribué MPI pour la dynamique de systèmes particulaires ». In : (2018) (cf. p. 12, 219).

[Pou20] Jean Pourroy. *Performance Modelisation*. 2020. URL : Liens (cf. p. 11).

[PP84] Mark S. Papamarcos et Janak H. Patel. « A low-overhead coherence solution for multiprocessors with private cache memories ». In : *Proceedings of the 11th annual international symposium on Computer architecture - ISCA '84*. T. 12. 3. ACM. 1984, p. 348–354. URL : Liens (cf. p. 271).

[PT13] John SHALF PARKERe et Li TANG. « On the role of co-design in high performance computing ». In : *Transition of HPC Towards Exascale Computing* 24 (2013), p. 141 (cf. p. 73).

[Put14] Bertrand Putigny. « Benchmark-driven Approaches to Performance Modeling of Multi-Core Architectures ». Thèse de doct. Université de Bordeaux, 2014. URL : Liens (cf. p. 265).

[Puz12] Nikola Puzovic. « Mont-Blanc : Towards energy-efficient HPC systems ». In : *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, p. 307–308 (cf. p. 173).

[PWW09] David Patterson, Samuel WIlliams et Andrew Waterman. « Roofline : An Insightful Visual Performance Model for Multicore Architectures ». In : *Communications of the ACM* (2009), p. 65–76. URL : Liens (cf. p. 107).

[Reb18] Anna Rebelles. *L'appli Ma P'tite poubelle, challenge zéro déchet à Barcelone – Data Metropole de Grenoble*. 2018. URL : Liens (cf. p. 44).

[Rei05] James Reinders. « VTune performance analyzer essentials ». In : *Intel Press* (2005) (cf. p. 168).

[Ric08] Thibault Ricart. *Etude de nano-systèmes électro-mécaniques (NEMS) à base de nanotubes de carbone pour applications hyperfréquences*. 2008 (cf. p. 68).

[Rob11] Robert D. *Intel PMT*. 2011. URL : Liens (cf. p. 298).

[Rod] Jorge Rodríguez. « Performance Analysis of Alya on a Tier-0 Machine using Extrae ». In : () (cf. p. 9, 105, 168, 189, 210).

[Röh+15] Thomas Röhl et al. « Overhead Analysis of Performance Counter Measurements ». In : *Proceedings of the International Conference on Parallel Processing Workshops* 2015-May.September (2015), p. 176–185 (cf. p. 299).

[Roj97] Raul Rojas. « The Architecture of the Z1 and Z3 ». In : *IEEE Annals of the History of Computing* 19.2 (1997), p. 5. URL : Liens (cf. p. 41).

[RS11] V Radhakishan et S Selvakumar. « Prevention of man-in-the-middle attacks using ID based signatures ». In : *2011 Second International Conference on Networking and Distributed Computing*. IEEE, 2011, p. 165–169 (cf. p. 84).

[Rup15] Karl Rupp. « 40 Years of Microprocessor Trend Data ». In : (2015). URL : Liens (cf. p. 4, 39, 243).

[Ryu18] RIKEN Junnichiro Makino Ryutaro Himeno Toshikazu Ebisuzaki. *The development of the application software for PEZY SC2 many-core processors*. Rapp. tech. 2018. URL : Liens (cf. p. 76).

[Sag16] Christian Saguez. « Calcul intensif et simulation numérique ». In : *Annales des Mines-Réalités industrielles*. 4. FFE, 2016, p. 32–36 (cf. p. 20).

[Sal18] J Saltz. « Big Data and Extreme-Scale computing : pathways to convergence ». In : Icl (2018). URL : Liens (cf. p. 45).

[Sch97] Robert R Schaller. « Moore's law : past, present and future ». In : *IEEE spectrum* 34.6 (1997), p. 52–59 (cf. p. 231).

[SDM10] John Shalf, Sudip Dosanjh et John Morrison. « Exascale computing technology challenges ». In : *International Conference on High Performance Computing for Computational Science*. Springer, 2010, p. 1–25. URL : Liens (cf. p. 47).

[Ses+17] Vivek Seshadri et al. « Ambit : In-memory accelerator for bulk bitwise operations using commodity DRAM technology ». In : *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, p. 273–287 (cf. p. 62).

[SGM19] Emma Strubell, Ananya Ganesh et Andrew McCallum. « Energy and Policy Considerations for Deep Learning in {NLP} ». In : *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy : Association for Computational Linguistics, juil. 2019, p. 3645–3650. URL : Liens (cf. p. 49).

[Sha+09] Sameh Sharkawi et al. « Performance projection of HPC applications using SPEC CFP2006 benchmarks ». In : *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, p. 1–12 (cf. p. 90).

[Shy+05] Alex Shye et al. « Analysis of path profiling information generated with performance monitoring hardware ». In : *9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*. IEEE, 2005, p. 34–43 (cf. p. 97).

[Shy+08] Alex Shye et al. « Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction ». In : *ACM SIGARCH Computer Architecture News*. T. 36. 3. IEEE Computer Society, 2008, p. 427–438 (cf. p. 97).

[SI02] Carl Staelin et Hewlett-Packardlaboratories Israel. « Lmbench3 : Measuring Scalability ». In : (2002) (cf. p. 91, 196).

[Sie69] C Sie. *Memory devices using bistable resistivity in amorphous As-Te-Ge films*. 1969 (cf. p. 68).

[Sim16] Evangelos Simoudis. *Insightful applications : The next inflection in big data*. 2016. URL : Liens (cf. p. 45).

[Sin+19] Gagandeep Singh et al. « Near-memory computing : Past, present, and future ». In : *Microprocessors and Microsystems* 71 (2019), p. 102868 (cf. p. 61).

[SL05] Herb Sutter et James Larus. « Software and the concurrency revolution ». In : *Queue* 3.7 (2005), p. 54. URL : Liens (cf. p. 42).

[SM06] Sameer S. Shende et Allen D. Malony. « The TAU parallel performance system ». In : *International Journal of High Performance Computing Applications* 20.2 (mai 2006), p. 287–311. URL : Liens (cf. p. 189).

[SMA14] C Sayin, Masteroppgave Master et Ects Avdeling. « Obfuscating Malware through Cache Memory Architecture Features ». Thèse de doct. 2014. URL : Liens (cf. p. 272).

[SMF11] Thomas Schaffter, Daniel Marbach et Dario Floreano. « GeneNetWeaver : in silico benchmark generation and performance profiling of network inference methods ». In : *Bioinformatics* 27.16 (2011), p. 2263–2270 (cf. p. 88).

[SMM17] Manuel Selva, Lionel Morel et Kevin Marquet. « Numap : A portable library for low-level memory profiling ». In : *Proceedings - 2016 16th International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation, SAMOS 2016* (2017), p. 55–62 (cf. p. 168, 169, 290).

[Spr02] Brinkley Sprunt. « The basics of performance-monitoring hardware ». In : *IEEE Micro* 22.4 (2002), p. 64–71. URL : Liens (cf. p. 98).

[SS20] Micholas Smith et Jeremy C Smith. « Repurposing Therapeutics for COVID-19 : Supercomputer-Based Docking to the SARS-CoV-2 Viral Spike Protein and Viral Spike Protein-Human ACE2 Interface ». In : (2020). URL : Liens (cf. p. 18).

[SS95] Rafael H. Saavedra et Alan Jay Smith. « Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes ». In : *IEEE Transactions on Computers* 44.10 (1995), p. 1223–1235. URL : Liens (cf. p. 8, 93, 120).

[Sta03] William Stallings. *Organisation et architecture de l'ordinateur*. Pearson Education, 2003 (cf. p. 250, 251).

[Sta05] Carl Staelin. « Lmbench : An extensible micro-benchmark suite ». In : *Software - Practice and Experience* 35.11 (2005), p. 1079–1105. URL : Liens (cf. p. 7, 88, 91).

[Sta13] JEDEC Standard. « High bandwidth memory (hbm) dram ». In : *JESD235* (2013) (cf. p. 63).

[STM05] Mile Stojcev, Teufik Toki et Ivan Milentijevi. « The limits of semiconductor technology and oncoming challenges in computer microarchitectures and architectures ». In : *Facta universitatis - series : Electronics and Energetics* 17 (jan. 2005), p. 285–312 (cf. p. 232).

[Sto87] Harold S Stone. *High-performance Computer Architecture*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1987 (cf. p. 269).

[Str+08] Dmitri B Strukov et al. « The missing memristor found ». In : *nature* 453.7191 (2008), p. 80–83 (cf. p. 62).

[Str18] Erich Strohmaier. *Highlights of the 51st TOP500 List*. Rapp. tech. Top500, 2018. URL : Liens (cf. p. 54, 77).

[Sut05] H Sutter. « The free lunch is over : A fundamental turn toward concurrency in software ». In : *Dr. Dobb's Journal* (2005), p. 1–9. URL : Liens (cf. p. 3).

[SW92] Arthur M Sherman et Lonnie S Walling. *Method for executing overlays in an expanded memory data processing system.* 1992 (cf. p. 273).

[Sze+17] Vivienne Sze et al. « Efficient processing of deep neural networks : A tutorial and survey ». In : *Proceedings of the IEEE* 105.12 (2017), p. 2295–2329 (cf. p. 57, 74).

[SZS07] Horst Simon, Thomas Zacharia et Rick Stevens. « Modeling and simulation at the exascale for energy and the environment ». In : *Department of Energy Technical Report* (2007) (cf. p. 44).

[Tan03] Andrew S Tanenbaum. *Structured computer organization.* T. 10. 3-4. Pearson Education India, 2003, p. 237 (cf. p. 228, 233, 251).

[Tan08] Andrew Tanenbaum. *Systemes d'exploitation.* Paris : Pearson Education, 2008 (cf. p. 273, 275, 279).

[Tan76] Calvin K Tang. « Cache system design in the tightly coupled multiprocessor system ». In : *Proceedings of the June 7-10, 1976, national computer conference and exposition.* ACM. 1976, p. 749–753 (cf. p. 271).

[THW10] Jan Treibig, Georg Hager et Gerhard Wellein. « LIKWID : A lightweight performance-oriented tool suite for x86 multicore environments ». In : *Proceedings of the International Conference on Parallel Processing Workshops* December 2016 (2010), p. 207–216 (cf. p. 105, 168).

[THW12] Jan Treibig, Georg Hager et Gerhard Wellein. « Likwid-bench : An extensible microbenchmarking platform for x86 multicore compute nodes ». In : *Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing 2011.* Springer, 2012, p. 27–36 (cf. p. 43).

[TW17] Thomas N Theis et H-S Philip Wong. « The end of moore's law : A new beginning for information technology ». In : *Computing in Science & Engineering* 19.2 (2017), p. 41 (cf. p. 232).

[Val16] Sébastien Valat. « Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance ». Thèse de doct. Juil. 2016. URL : Liens (cf. p. 30).

[Vas+14] Nicolas Vasilache et al. « Fast convolutional nets with fbfft : A GPU performance evaluation ». In : *arXiv preprint arXiv :1412.7580* (2014) (cf. p. 57).

[Vie96] Science et Vie. *Les Cahiers de Science et Vie, Hors série N36 : Qui a inventé l'ordinateur ?* 1996 (cf. p. 234).

[Wan+01] P H Wang et al. « Register renaming and scheduling for dynamic execution of predicated code ». In : *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture.* Jan. 2001, p. 15–25 (cf. p. 245).

[WD98] R.C. Whaley et J.J. Dongarra. « Automatically Tuned Linear Algebra Software ». In : *Proceedings of the IEEE/ACM SC98 Conference*. IEEE. 1998, p. 38–38. URL : Liens (cf. p. 88).

[Wea13] Vincent M Weaver. « Linux perf_event Features and Overhead ». In : *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath* April (2013), p. 80 (cf. p. 9, 102, 168).

[Wea15] Vincent M. Weaver. « Self-monitoring overhead of the Linux perf- event performance counter interface ». In : *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software* (2015), p. 102–111 (cf. p. 299).

[Wik] Wikichip. *Skylake (client) - Microarchitectures - Intel - WikiChip*. URL : Liens (cf. p. 278).

[wik] wikichip. *Xeon Gold 6148 - Intel - WikiChip*. URL : Liens (cf. p. 194).

[Wik19a] Wikibooks. *Fonctionnement d'un ordinateur*. 2019. URL : Liens (cf. p. 226).

[Wik19b] Wikipedia. *Cache inclusion policy*. Mar. 2019. URL : Liens (cf. p. 266).

[Wik19c] Wikipedia. *Cache replacement policies*. Avr. 2019. URL : Liens (cf. p. 268).

[Wik19d] Wikipedia. *Fonction logique*. 2019. URL : Liens (cf. p. 227).

[Wik19e] Wikipedia. *Transistor Count*. 2019. URL : Liens (cf. p. 232).

[Wil01] Maurice V Wilkes. « The memory gap and the future of high performance memories ». In : *ACM SIGARCH Computer Architecture News* 29.1 (2001), p. 2–7 (cf. p. 3, 60).

[Wil08] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. 2008 (cf. p. 196).

[Wil65] M. V. Wilkes. « Slave Memories and Dynamic Storage Allocation ». In : *IEEE Transactions on Electronic Computers* EC-14.2 (1965), p. 270–271 (cf. p. 264).

[WM95] Wm A Wulf et Sally A McKee. « Hitting the memory wall : implications of the obvious ». In : *ACM SIGARCH computer architecture news* 23.1 (1995), p. 20–24 (cf. p. 39, 255).

[Wol57] Jacques Wolff. « Entrepreneurs et firmes. Ford et Renault de leurs débuts à 1914 ». In : *Revue économique* 8.2 (1957), p. 297–323 (cf. p. 247).

[Won+15] David C Wong et al. « Vp3 : A vectorization potential performance prototype ». In : *Workshop on Programming Models for SIMD/Vector Processing*. 2015 (cf. p. 96).

[Won02] David Wonnacott. « Achieving scalable locality with time skewing ». In : *International Journal of Parallel Programming* 30.3 (2002), p. 181–221 (cf. p. 57).

[Xue12] Jingling Xue. *Loop tiling for parallelism*. T. 575. Springer Science & Business Media, 2012 (cf. p. 57).

[Yos+15] T Yoshida et al. « Sparc64 XIfx : Fujitsu's Next-Generation Processor for High-Performance Computing ». In : *IEEE Micro* 35.02 (mar. 2015), p. 6–14 (cf. p. 55).

[Yos18] Toshio Yoshida. « Fujitsu high performance cpu for the post-k computer ». In : *Hot Chips 30 Symposium (HCS), Series Hot Chips*. T. 18. 2018 (cf. p. 55).

[YPS04] Kamen Yotov, Keshav Pingali et Paul Stodghill. « X-Ray : Automatic Measurement of Hardware Parameters ». In : *Framework* (2004), p. 1–22 (cf. p. 92).

[YPS05] Kamen Yotov, Keshav Pingali et Paul Stodghill. « Automatic measurement of memory hierarchy parameters ». In : *ACM SIGMETRICS Performance Evaluation Review* 33.1 (2005), p. 181. URL : Liens (cf. p. 7, 92, 93, 120).

[Yun+13] Heechul Yun et al. « Memguard : Memory bandwidth reservation system for efficient performance isolation in multi-core platforms ». In : *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, p. 55–64 (cf. p. 168).

[ZDS09] Xiao Zhang, Sandhya Dwarkadas et Kai Shen. « Towards practical page coloring-based multicore cache management ». In : *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, p. 89–102 (cf. p. 139).

[Zha+17] Yaoxue Zhang et al. « A survey on emerging computing paradigms for big data ». In : *Chinese Journal of Electronics* 26.1 (2017), p. 1–12 (cf. p. 43).

[ZWV16] Jintao Zhang, Zhuo Wang et Naveen Verma. « A machine-learning classifier implemented in a standard 6T SRAM array ». In : *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. IEEE, 2016, p. 1–2 (cf. p. 62).

université PARIS-SACLAY
ÉCOLE DOCTORALE
de mathématiques
Hadamard (EDMH)

Titre : Calcul Haute Performance : Caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs

Mots clés : HPC ; Exascale ; Supercalculateur ; Performance ; Optimisation ; Benchmark

Résumé : Le domaine du Calcul Haute Performance (HPC) participent activement à l'amélioration des connaissances scientifiques et les utilisateurs ont besoin d'architectures toujours plus puissantes pour réaliser des simulations numériques plus rapidement. Dans ce travail de thèse, nous discutons des différents challenges à relever pour l'élaboration des nouvelles générations de supercalculateurs. Nous présentons alors certaines technologies émergentes permettant leur développement. Notre contribution consiste au développement d'une suite de benchmark et d'outils d'analyse de performance. Les premiers ont pour objectifs de caractériser finement certaines parties de la microarchitecture. Les deuxièmes permettent d'étudier la performance des applications. Nous proposons une méthodologie en 5 étapes permettant d'identifier et de caractériser ces nouvelles plateformes, de modéliser les performances d'une application, et enfin de porter son code sur l'architecture choisie. Enfin, nous montrons comment les outils permettent d'accompagner les développeurs pour extraire le maximum des performances d'une architecture.

Title : High-Performance Computing : Architecture characterization and application optimization for future generations of supercomputers

Keywords : HPC ; Exascale ; Supercomputer ; Performance ; Optimization ; Benchmark

Abstract : The High Performance Computing (HPC) field is actively involved in the improvement of scientific knowledge and users always need more and more powerful architectures to perform numerical simulations faster. In this thesis work, we discuss the different challenges to be addressed for the development of new generations of supercomputers. We then present some of the emerging technologies enabling their development. Our contribution consists in the development of a suite of benchmarks and performance analysis tools. The first aim is to finely characterize certain parts of the microarchitecture. The second allow to study the performance of applications. We propose a 5-step methodology to identify and characterize these new architectures, to model the performance of an application, and finally to port its code. Finally, we show how the tools can help developers to extract the maximum performance from an architecture.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France