Étude de transformations et d'optimisations de code parallèle statique ou dynamique pour architecture "many-core"

Camille Gallet

UPMC
SORBONNE UNIVERSITÉS

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Camille GALLET

CEA, DAM, DIF, F-91297 Arpajon, France

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Étude de transformations et d'optimisations de code parallèle statique ou dynamique pour architecture « many-core »

Soutenue le 03 octobre 2016

Devant le jury composé de :

M. Albert Cohen	Directeur de thèse
M. William Jalby	Rapporteur
M. Henri-Pierre Charles	Rapporteur
Mme. Pascale Rossé-Laurent	Examinatrice
M. Lionel Lacassagne	Examinateur
M. Patrick Carribault	Examinateur

Résumé

L'évolution des supercalculateurs, de leur origine dans les années 60 jusqu'à nos jours, a fait face à 3 révolutions : (i) l'arrivée des transistors pour remplacer les triodes, (ii) l'apparition des calculs vectoriels, et (iii) l'organisation en grappe (clusters). Ces derniers se composent actuellement de processeurs standards qui ont profité de l'accroissement de leur puissance de calcul via une augmentation de la fréquence, la multiplication des cœurs sur la puce et l'élargissement des unités de calcul (jeu d'instructions SIMD). Un exemple récent comportant un grand nombre de cœurs et des unités vectorielles larges (512 bits) est le co-proceseur Intel Xeon Phi. Pour maximiser les performances de calcul sur ces puces en exploitant aux mieux ces instructions SIMD, il est nécessaire de réorganiser le corps des nids de boucles en tenant compte des aspects irréguliers (flot de contrôle et flot de données). Dans ce but, cette thèse propose d'étendre la transformation nommée Deep Jam pour extraire de la régularité d'un code irrégulier et ainsi faciliter la vectorisation. Ce document présente notre extension et son application sur une mini-application d'hydrodynamique multi-matériaux HydroMM. Ces travaux montrent ainsi qu'il est possible d'obtenir un gain de performances significatif sur des codes irréguliers.

Table des matières

1 Introduction **1**
- 1.1 Contexte historique . . . 1
- 1.2 Accroissement des performances des architectures . . . 3
  - 1.2.1 Augmentation de la fréquence . . . 5
  - 1.2.2 Apparition des processeurs à plusieurs cœurs . . . 7
  - 1.2.3 Vers les processeurs manycore et ressources dédiées . . . 8
  - 1.2.4 Élargissement des unités de calcul . . . 9
  - 1.2.5 Exemple du processeur Intel Xeon Phi . . . 11
- 1.3 Exploitation du parallélisme SIMD . . . 13
  - 1.3.1 Historique de la vectorisation . . . 13
  - 1.3.2 Exemple de vectorisation manuelle . . . 14
    - Vectorisation automatique . . . 17
    - Vectorisation guidée . . . 17
    - Vectorisation manuelle . . . 18
    - Résultats . . . 19
- 1.4 Synthèse . . . 19

2 État de l'art sur la vectorisation de code irrégulier **21**
- 2.1 Vectorisation explicite . . . 22
- 2.2 Vectorisation automatique de boucles . . . 24
  - 2.2.1 Parallélisme vectoriel d'un niveau de boucle . . . 24
    - Support d'un seul niveau de flot de contrôle . . . 26
    - Gestion de flot de contrôle imbriqué . . . 29
  - 2.2.2 Parallélisme vectoriel d'un nid de boucle . . . 30
    - Vectorisation dans un nid de boucle parfait uniquement . . . 30
    - Prise en charge des nids de boucles non parfaits . . . 31
  - 2.2.3 Vectorisation d'une fonction, SPMD . . . 33
- 2.3 Transformation pour faciliter la vectorisation . . . 35
- 2.4 Conclusion . . . 39

3 Extension de la transformation Deep Jam pour la vectorisation **41**
- 3.1 Catalogue de transformations . . . 41
  - 3.1.1 Hoisting . . . 41
  - 3.1.2 Spécialisation . . . 42
  - 3.1.3 Strip Mining . . . 42
  - 3.1.4 Tiling . . . 42
  - 3.1.5 Fusion . . . 43

3.1.6 Fission . . . 43
3.1.7 Spliting . . . 43
3.1.8 Interchange . . . 44
3.1.9 Peeling . . . 44
3.2 Algorithme du Deep Jam étendu . . . 44
3.2.1 Exemple préliminaire d'application . . . 44
3.2.2 Algorithme du Deep Jam étendu à la vectorisation de code irrégulier . . . 45
3.2.3 Exemple d'impact du support du Single Instruction Multiple Data (SIMD) sur deux architectures . . . 46
3.2.4 Exemple de l'impact du choix du compilateur . . . 46
3.2.5 Synthèse des besoins . . . 47
3.3 Des microbenchmarks pour expliciter les oracles . . . 48
3.3.1 Microbenchmark n° 1 : for if / for if else . . . 48
3.3.2 Micro-benchmark n° 2 : for if if . . . 49
3.3.3 Micro-benchmark n° 3 : for if for / for if for else for . . . 51
3.3.4 Réflexion pour le flot de données . . . 55
3.3.5 Résumé et conclusions . . . 57
3.4 Conclusions, limites . . . 57

4 Application manuelle du Deep Jam adapté sur un code de calcul 59
4.1 Le benchmark HydroMM . . . 59
4.2 Vectorisation du benchmark . . . 62
4.2.1 Première étape, les points chauds . . . 62
4.2.2 Flot de contrôle . . . 64
4.2.3 Une première régularisation du flot de données . . . 68
4.3 Résultats . . . 70
4.3.1 Vectorisation automatique par le compilateur . . . 70
4.3.2 SIMD et flot de contrôle . . . 70
4.3.3 Résultats pour le flot de données . . . 72
4.4 Synthèse . . . 73

5 Conclusion et perspectives 75
5.1 Conclusion . . . 75
5.2 Travaux futurs . . . 76
5.2.1 Court terme . . . 76
5.2.2 Long terme . . . 78

Glossary 79

Bibliographie 79

Chapitre 1

Introduction

Ce chapitre présente une introduction générale à cette thèse. Pour mieux comprendre la problématique posée par nos travaux de recherche, cette première partie introduit tout d'abord le contexte général du calcul haute performance. Après cette description historique, nous nous concentrerons sur l'évolution des architectures de supercalculateurs et, plus précisément, des noeuds de calcul pour comprendre l'arrivée massive d'unités vectorielles dans les processeurs modernes. Enfin, ce chapitre se termine avec l'introduction de la problématique de cette thèse : la difficulté d'exploiter les unités vectorielles sur les processeurs hautes performances.

1.1 Contexte historique

Cette première section présente un résumé de l'évolution des supercalculateurs pour le Calcul Haute Performance (HPC) [1].

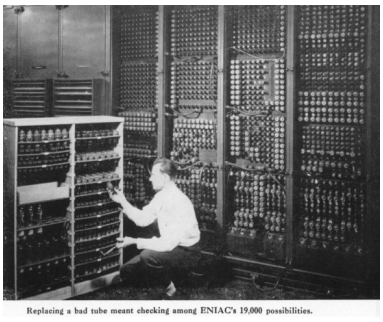


FIGURE 1.1 – Ordinateur Eniac

Un des tout premiers ordinateurs dédiés pour le calcul fut l'Eniac (Electronic Numerical Integrator and Computer). Conçu en 1943, il représente le premier ordinateur exclusivement électronique. La Figure 1.1 présente une illustration de cette approche.

Plus de vingt ans plus tard, l'entreprise CDC (Control Data Corporation) sort une série d'ordinateurs dont le CDC 6600 développé et vendu en 1964. Cette machine est ainsi considérée comme le premier supercalculateur avec une puissance de calcul se situant entre 512 et 1024 KFlops, c'est-à-dire une capacité de calcul entre 512 et 1024 milliers d'opérations flottantes par seconde. La Figure 1.2 illustre cette machine.

1. http://www.math-cs.gordon.edu/courses/cps343/presentations/HPC_History.pdf

FIGURE 1.2 – Ordinateur CDC 6600

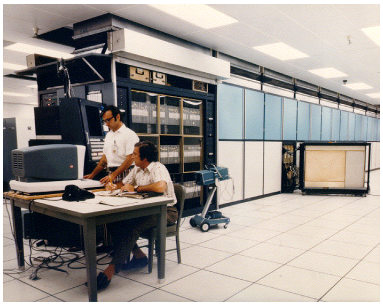


FIGURE 1.3 – Illiac-IV

Quelques années plus tard est apparu un nouveau supercalculateur du nom de ILLIAC-IV (Illinois Automatic Computer). Il fut construit à partir de 1966 pour être déployé au début des années 1970. Il fut opérationnel à partir de cette date jusqu'en 1976. Sa puissance crête devait normalement se situer au niveau de 1 GFlops (gigaflops, c'est-à-dire un milliard d'opérations flottantes par seconde), mais au final il atteignit les 200 MFlops (mégaflops, c'est-à-dire 200 millions d'opérations flottantes par seconde). La Figure 1.3 présente une photographie de ce supercalculateur.

FIGURE 1.4 – Cray 1

Au même moment, un nouveau type de machines pour le HPC apparut : les supercalculateurs vectoriels. Les processeurs vectoriels sur ces machines sont conçus pour utiliser un pipeline pour réaliser une opération sur un nombre flottant sur une grande quantité de données. Les premiers à fonctionner correctement et avoir un franc succès furent ceux de l'entreprise de Seymour Cray (ancien employé de l'entreprise CDC) avec le Cray 1 en 1976 (133 MFlops soit 133 millions d'opérations à la seconde). La Figure 1.4 montre une photographie du premier Cray. Après cela d'autres versions furent conçues : le Cray X-MP en 1982 (200 MFlops) puis le

Cray Y-MP en 1988 pour 333 MFlops jusqu'au Cray XC30-AC en 2013, machine petaflopique c'est-à-dire capable d'exécuter un million de milliards (soit 10^{15}) d'opérations par seconde.

À la fin des années 1980, une machine précurseure, l'Intel iPSC Hypercube, ouvre en 1985 une nouvelle ère : les clusters. Cette machine, dont nous pouvons voir une photo sur la Figure 1.5, a entre 32 et 128 nœuds avec 512 Ko de RAM chacun pour un total de 80286 processeurs et une puissance d'environs 2 MFlops. En effet, un cluster veut simplement dire que le supercalculateur est constitué d'ordinateurs reliés entre eux avec comme avantage une baisse des coûts et une maintenance plus aisée. Pour illustrer les prix, le cluster Beowulf qui atteignit 1GFlops n'a coûté que 50 000$ en 1994.

FIGURE 1.5 – Intel iPSC source : `https://www.flickr.com/photos/carrierdetect/3599265588/`

Comme nous pouvons le voir sur la Figure 1.6 l'arrivée des clusters au départ a permis une forte diversification des architectures utilisées pour construire les supercalculateurs. Sur ce graphe, chaque zone coloriée représente une architecture et son épaisseur indique son taux d'utilisation par rapport aux autres. Nous voyons aussi sur cette figure que ces dernières années l'architecture la plus courante est celle utilisée sur les ordinateurs et serveurs standards : x86-64 d'Intel. Encore une fois, c'est toujours dans l'optique de réduire le coût de mise en place et de maintenance.

Pour conclure cette section, voici un récapitulatif visible sur la Figure 1.7. Celle-ci montre l'accroissement de la puissance crête des supercalculateurs des années 50 aux années 2000. Il est important de noter que l'axe des ordonnées (Flops) est logarithmique ce qui montre une évolution exponentielle qui n'a pas faibli. L'amélioration des performances que nous avons vue pour les supercalculateurs a été possible grâce à plusieurs facteurs que nous allons détailler dans la section suivante.

1.2 Accroissement des performances des architectures

Les premières technologies des ordinateurs reposaient sur des tubes électroniques (triodes) pour encoder les traitements en binaire. La Figure 1.8 illustre la composition et le fonctionnement d'un tel tube. Ce dernier contient une cathode et une plaque séparées par une grille, le tout suspendu dans un tube en verre sous vide.

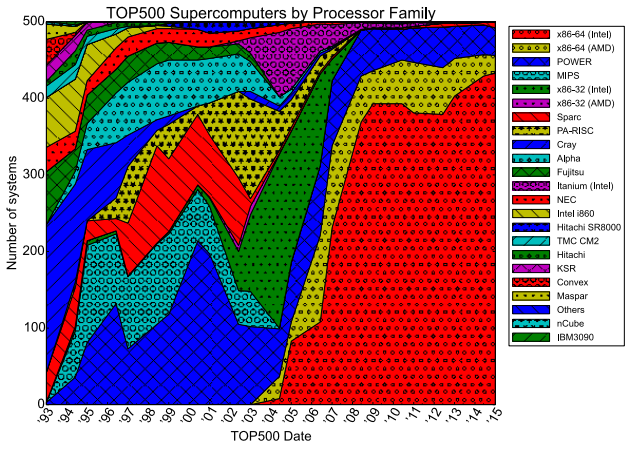


FIGURE 1.6 – Diversification des architectures des supercalculateurs

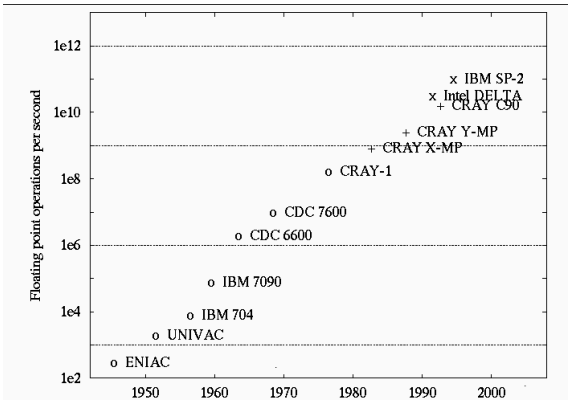


FIGURE 1.7 – Évolution de la puissance crête

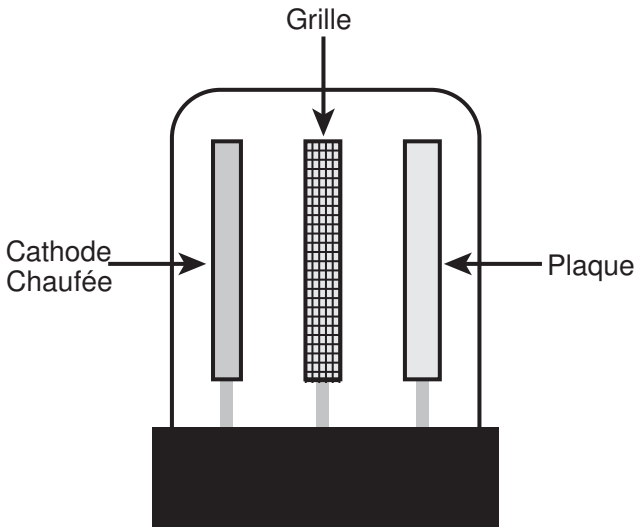


FIGURE 1.8 – Tube électronique (triode) [47]

Grâce à ce dispositif, il était possible d'encoder la notion de 0 et 1 binaire de la façon suivante : la cathode est tout d'abord chauffée, ce qui lui fait émettre des électrons attirés par la plaque. La grille a alors le rôle de

pouvoir contrôler le flot d'électrons. Il existe alors deux cas : (i) quand la grille devient chargée négativement, le flot d'électrons retourne vers la cathode et (ii) quand la grille devient positive, c'est la plaque qui reçoit ce flot [47]. Ce phénomène permet ainsi d'encoder un signal électrique au format binaire.

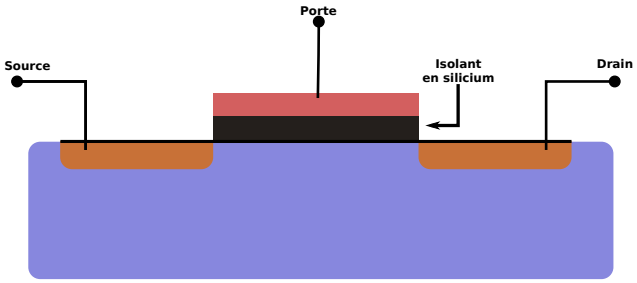


FIGURE 1.9 – Transistor MOSFET

Le problème principal de l'approche précédente est l'efficacité et la rapidité avec laquelle le mécanisme permet de passer d'une valeur binaire à une autre. C'est pour cette raison que le transistor fera son apparition en 1947. Cette nouvelle approche s'imposera alors petit à petit pour la conception d'ordinateurs. Ces transistors fonctionnent de la manière suivante : le courant ne peut passer de la source au drain que si la porte est *ouverte* (activée). La Figure 1.9 illustre ce mécanisme. En jouant ainsi sur la porte, il est possible de procéder à un encodage binaire. Les transistors sont toujours à la base des processeurs modernes appelés CPU (Central Processing Unit). Le premier ordinateur totalement à base de transistors fut le CDC 1604 [2].

Mais il devient alors nécessaire d'organiser ces transistors entre eux et de procéder à des optimisations pour augmenter les performances des processeurs, c'est-à-dire accroitre le nombre d'opérations par seconde qu'ils sont capables d'effectuer. Une première approche consiste à augmenter la fréquence du processeur cible. Cette fréquence caractérise ainsi la durée d'un cycle d'horloge interne du processeur, guidant ainsi le nombre de fois que la porte de chaque transistor peut s'ouvrir ou se fermer.

1.2.1 Augmentation de la fréquence

La première solution, comme évoquée dans le paragraphe précédent, consiste à augmenter la fréquence à laquelle les transistors s'activent et se referment. Ceci a naturellement comme premier effet d'accélérer les capacités de traitement du processeur. En effet, doubler la fréquence revient alors à exécuter les mêmes instructions de calcul deux fois plus vite, doublant par la même occasion les performances générales du processeur. Cette tendance fut suivie pendant plusieurs décennies. En effet, la Figure 1.10 illustre l'évolution des performances des processeurs de 1978 à 2008 par rapport à un processeur cible (VAX-11/780). Cette courbe intègre également la fréquence de ces processeurs : de 0,05 GHz en 1991, un plafond fut atteint vers les années 2000 autour de quelques GHz. Ainsi, l'augmentation de la fréquence des processeurs s'est progressivement ralentie, pour stagner depuis les années 2010. En effet, plusieurs problèmes sont apparus expliquant cette évolution : la chaleur émise et la consommation électrique.

La Figure 1.11 illustre les problèmes majeurs liés à l'élévation de la fréquence. Ce graphique présente l'évolution, de 1982 à 2007, de la fréquence des processeurs (axe des ordonnées de gauche) ainsi que la consommation électrique de la puce (axe des ordonnées de droite). Le problème identifié à la Figure 1.10 s'illustre également dans ces courbes : la fréquence n'a cessé d'augmenter pendant plusieurs décennies, de concert avec la consommation, jusqu'à stagner à partir du milieu des années 2000 à cause, en majeure partie, de cette consommation. Ainsi, Intel a fait un choix de revenir en arrière en termes de conception de processeurs

2. CDC 1604 : `http://histoire.info.online.fr/super.html`

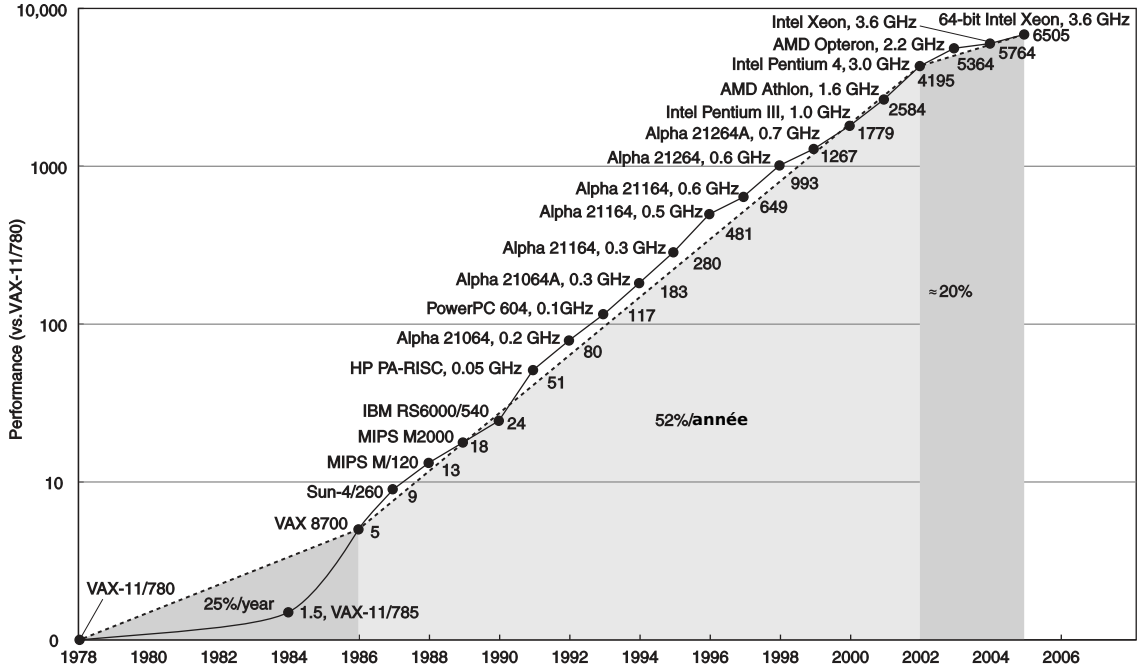


FIGURE 1.10 – Évolution des performances des processeurs

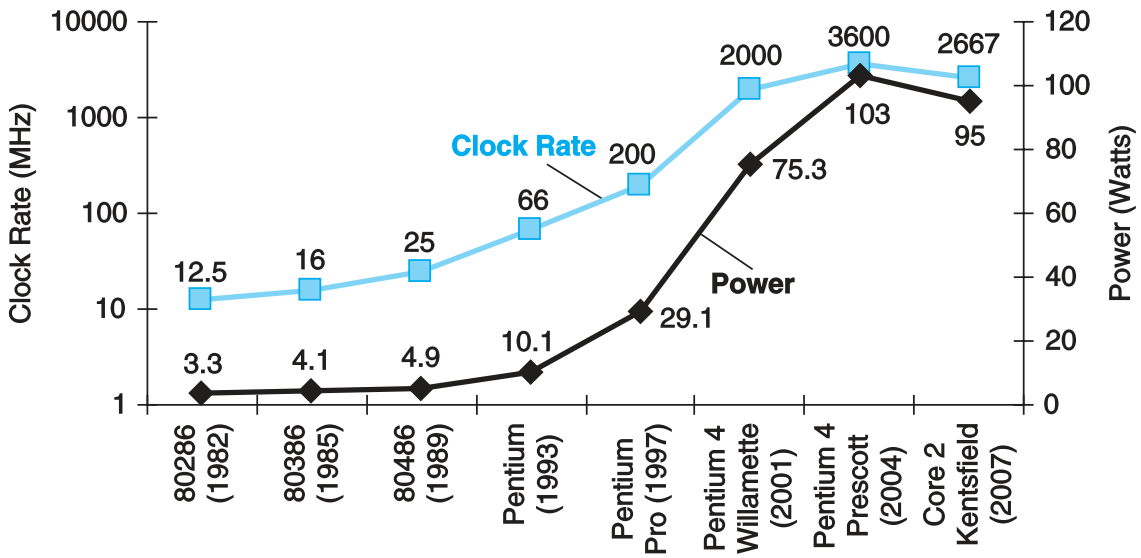


FIGURE 1.11 – Évolution de la fréquence et consommation électrique [a]

a. Source : `http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf`

pour passer du Pentium 4 (Prescott) au Core 2, basé sur le schéma du Pentium 3. Cette bascule a permis de conserver une fréquence importante sans pour autant faire exploser la consommation électrique. Il est également important de souligner que l'élévation de la fréquence implique une élévation de température. La dissipation de cette chaleur participe également à la surconsommation électrique.

La Figure 1.12 présente un autre point de vue. En effet, sur ce graphe, les performances des processeurs sur

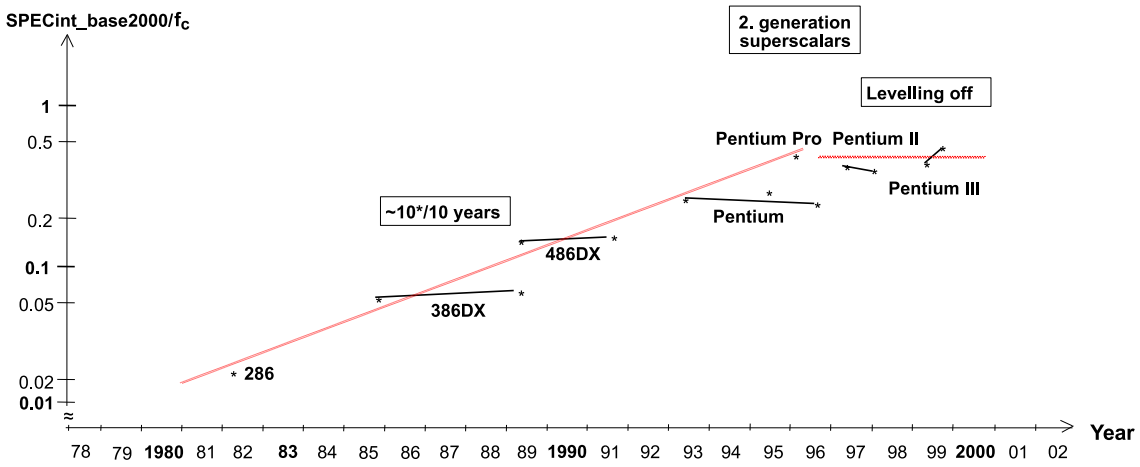


FIGURE 1.12 – Évolution de l'efficacité en fonction du temps source : http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf

la suite de benchmarks SPECint [3] sont présentées en fonction de l'année de sortie de ce dernier. Il est à noter que la notion de performances est ici relative à la fréquence du processeur cible. Ainsi, un autre phénomène forçant ce ralentissement est l'arrivée à stagnation de l'efficacité ; l'efficacité étant définie comme la façon dont un programme (code) utilise un processeur. Les segments distincts symbolisent des performances similaires en termes des résultats du benchmark SPECint_ base2000. L'évolution de cette efficacité est passée, bien entendu, par une montée en fréquence et l'ajout de matériel optimisant au sein même du processeur. L'exemple donné sur cette courbe est la notion de *superscalaire* qui est une technique permettant d'exécuter plusieurs instructions scalaires (agissant sur principalement une seule donnée) en même temps. Ainsi, la seconde génération de superscalaire a permis d'atteindre de meilleures performances vers le milieu des années 90.

Mais les techniques classiques d'augmentation des performances ont connu leurs limites dans les années 2000 (montée en fréquence, mécanisme superscalaire, exécution dans le désordre, exploitation du parallélisme entre les instructions à exécuter, ...). Donc, pour continuer à augmenter les performances, les architectes ont dû explorer une autre direction, profitant ainsi de la diminution de la finesse de gravure : l'augmentation du nombre de cœurs de calcul par processeur.

1.2.2 Apparition des processeurs à plusieurs cœurs

Comme nous l'avons évoqué dans la section précédente, l'augmentation de la fréquence et l'optimisation des différents composants d'un processeur (pour améliorer l'exécution des codes) ont atteint des limites dans les années 2000. Mais la technologie de conception et fabrication des transistors a continué à évoluer permettant ainsi de diminuer la finesse de gravure. Ceci avait pour effet immédiat de pouvoir placer plus de transistors sur une même surface. Il est alors important de se demander comment il est possible d'augmenter les performances d'un processeur en profitant de plus de place pour rajouter des transistors. Les architectes de processeurs ont alors décidé de dupliquer, non seulement les unités de calcul pour augmenter les performances crêtes (donc maximales) du processeur, mais aussi de dupliquer une bonne partie du mécanisme d'exécution pour laisser le système utiliser lui-même ces unités de calcul. C'est ainsi que sont nés les processeurs *multicœurs*.

3. https://www.spec.org/cpu2000/

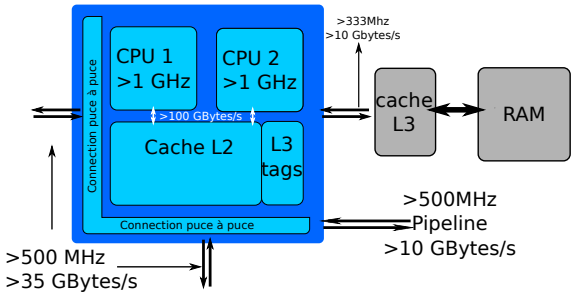


FIGURE 1.13 – Schéma IBM Power4

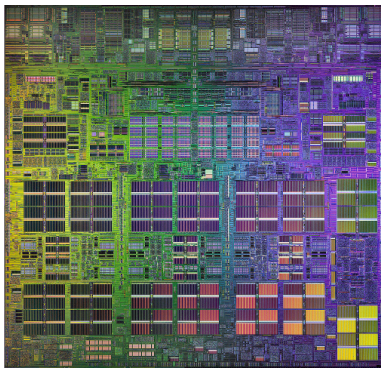


FIGURE 1.14 – Vue IBM Power4 [a]

a. source : http://pages.cs.wisc.edu/~cain/images/diephotos/

Par la suite, tous les CPUs sont devenus multicœurs avec un faible nombre de cœurs. Le premier processeur de ce type fut le Power4 d'IBM sorti en 2001 [4]. La Figure 1.13 présente une vue logique de ce processeur : il est composé de deux cœurs de calcul indépendants (nommé CPU1 et CPU2 sur le schéma) qui contiennent leur propre cache de niveau L1 privé. Le cache de niveau 2 (dénommé Cache L2 sur la figure) est partagé par les deux cœurs, de même que le cache de niveau 3 situé en dehors de la puce principale. La Figure 1.14 illustre la disposition des éléments sur ce processeur avec une vue physique. Il est alors possible de voir les deux cœurs de calcul en remarquant les symétries et réplications de composants sur la photo. Par conséquent, ce processeur permet, en théorie, d'atteindre des performances doubles par rapport à la même approche composée d'un seul cœur de calcul. En effet, les deux boites bleues CPU1 et CPU2 contiennent exactement les mêmes composants et le même nombre de transistors. Ainsi chaque cœur est capable d'exécuter son propre flot d'instructions (thread ou processus d'un point de vue système). Il est important de noter que, contrairement aux optimisations décrites dans la section précédente, il est à la charge de la pile logicielle d'exploiter tous les cœurs de calcul. En effet, l'augmentation de la fréquence, la gestion du parallélisme d'instructions, l'exécution simultanée de plusieurs instructions de calcul sont des techniques qui améliorent les performances de n'importe quel code sans nécessiter du travail du côté de la pile logicielle (compilateur, développeurs d'applications, système d'exploitation, ...). Ce n'est pas le cas de l'approche multicœur : le code doit avoir exprimé du parallélisme pour pouvoir utiliser tous les cœurs disponibles.

1.2.3 Vers les processeurs manycore et ressources dédiées

La mise en place progressive des processeurs multicœurs s'est faite dans le courant des années 2000. La Figure 1.15 présente une vision schématique et résumée de l'évolution du nombre de cœurs au sein des processeurs modernes. Sur la gauche, on voit apparaitre les processeurs tels qu'ils étaient jusqu'à la fin des années 90 : l'approche single-core (monocœurs). Les contraintes étaient alors, comme décrites dans les sections précédentes, la montée en puissance et l'efficacité sur des benchmarks et applications. Nous avons vu l'apparition nécessaire et logique des processeurs multicœurs qui représentent le milieu de la figure. Une fois encore cette approche a pour contrainte la puissance, car dupliquer des cœurs de calcul contenant beaucoup de logique (exécution dans le désordre, mécanisme superscalaire, ...) est très demandeur en consommation. De plus le problème du passage à l'échelle est crucial, car la pile logicielle est alors en charge d'exploiter efficacement tous les cœurs disponibles. Pour toutes ces raisons, les architectures des constructeurs Intel,

4. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

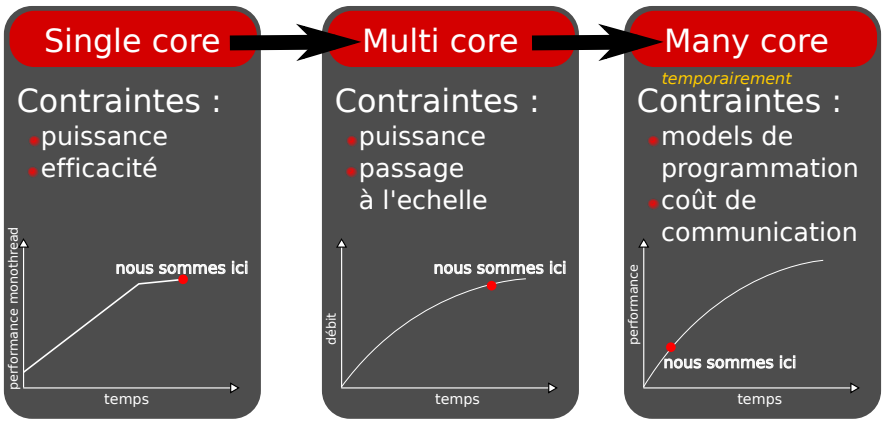


FIGURE 1.15 – Évolution des cœurs des processeurs [a]

a. http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck/3

AMD et NVIDIA ont été plus loin et ont proposé la notion de processeurs manycore (partie droite de la Figure 1.15). On peut résumer les propriétés de tels processeurs de la manière suivante :

— plus de cœurs de calcul et de threads eux-mêmes plus légers ;
— plus de capacité de calculs dédiés au calcul *data-parallel* haute performance
— plus de bande passante mémoire, moins de caches et de mémoire par cœur.

De par ces différences les architectures manycore nécessitent une focalisation sur des techniques de programmation et des paradigmes différents pour exploiter efficacement le parallélisme vectoriel avec les autres niveaux de parallélisme. Mais il existe plusieurs manières différentes d'appréhender la notion de processeurs manycore. Il est possible de jouer sur le nombre de cœurs et/ou sur la complexité de chacun de ces cœurs.

Une approche consistant à dupliquer le nombre de cœurs de façon intensive est celle prise par le constructeur de carte graphique Nvidia. En effet, les GPGPU [5] sont des processeurs « graphiques » dédiés pour le domaine du calcul haute performance. Pour comprendre leur fonctionnement, il faut étudier celui des puces sous-jacentes. De manière générale, les carte graphique (GPU) sont composés de très nombreux cœurs de calcul légers qui sont fortement adaptés pour le parallélisme de données [54, 55]. Si nous nous focalisons sur les GPU Nvidia, l'architecture se décompose comme suit : chaque kernel (portion de code à exécuter sur le GPU) s'exécute sur une grille de blocs par des « Stream Multiprocessor » (Stream Multiprocessor (SM)). Tout d'abord, comme il est visible sur la Figure 1.16 entre les SM ou entre les cœurs d'un SM nous avons du parallélisme de threads. Mais aussi nous pouvons aussi utiliser différents GPU en même temps (d'autant plus sur les GPGPU Nvidia récents comprenant plusieurs chipsets sur la même carte). Donc nous pouvons voir ainsi qu'il y a du parallélisme multi niveaux à exploiter.

1.2.4 Élargissement des unités de calcul

L'augmentation du nombre de cœurs de calcul n'est qu'une des pistes utilisées pour accroitre la performance crête des processeurs en gardant l'enveloppe thermique en dessous d'un seuil critique. Un autre mécanisme a également été développé dans ce but : l'élargissement des unités de calcul. Ainsi, pour étendre la capacité de calcul sans toucher ni à la fréquence ni au nombre de cœurs, les fabricants de puces ont rajouté des

5. http://gpgpu.org/

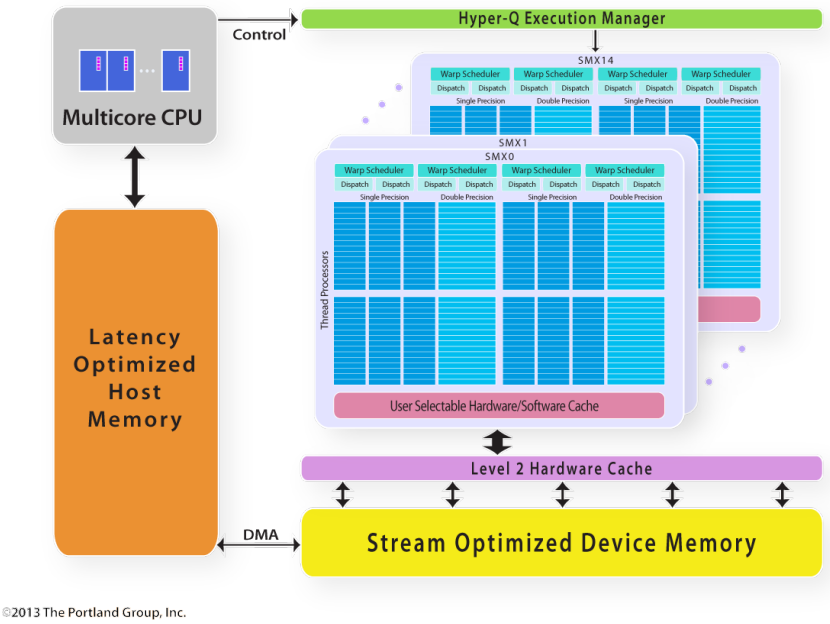


FIGURE 1.16 – Architecture d'un carte graphique dédiée au HPC (GPGPU) K80 de Nvidia

registres et des unités particuliers pour effectuer une opération sur plusieurs données indépendantes en une seule instruction. Par exemple, il est alors possible de créer un registre 128 bits contenant la concaténation de 4 scalaires de 32 bits. L'addition de deux de ces registres implique alors le passage par l'unité de calcul élargie qui effectue, en parallèle, l'addition des 4 éléments de ces vecteurs. Le résultat est ainsi un registre 128 bits contenant la concaténation du résultat des 4 additions 32 bits. On parle alors d'instruction SIMD (Single Instruction Multiple Data) ou d'instruction vectorielle (sur des registres et unités vectoriels).

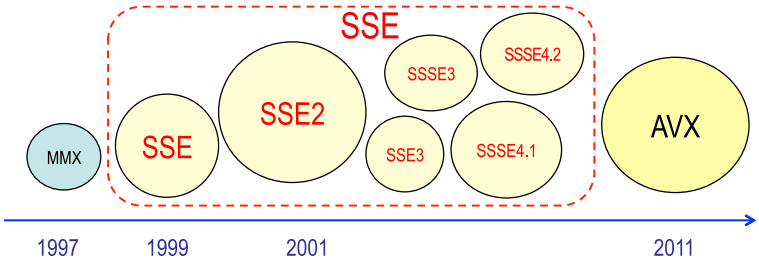


FIGURE 1.17 – Évolution des jeux d'instructions vectoriels Intel [a]

a. Source : `http://svmoore.pbworks.com/w/file/fetch/70583970/VectorOps.pdf`

Il est alors important de noter que cette technologie implique l'extension du jeu d'instructions par l'ajout de nouvelles instructions pour exploiter ces nouvelles unités et ces nouveaux registres. La Figure 1.17 présente la répartition et l'évolution de ces nouveaux jeux d'instructions au cours du temps avec leur nom respectif. Ainsi, la première implémentation de cette approche fut introduite avec le processeur Pentium d'Intel en 1993. À cette époque, elle se nomme alors MultiMedia eXtensions (MMX) et ses registres vectoriels dédiés comportent 64 bits, soit de quoi stocker 2 entiers de 32 bits simultanément. Par la suite, c'est avec le Pentium III qu'Intel dévoile en 1999 le Streaming SIMD Extensions (SSE) avec des registres vectoriels deux fois plus grands supportant cette fois-ci les calculs flottants en simple précision. Cette technologie a été étendue

du SSE2 (qui, en 2000, a introduit le support des calculs flottants en double précision) au SSE4-2 proposé sur les processeurs de type Nehalem. C'est en 2008 que Intel a dévoilé l'Advanced Vector eXtension (AVX) qui, une nouvelle fois, propose de doubler la taille des registres vectoriels pour atteindre un total de 256 bits. Récemment, en 2013, Intel a sorti une extension nommée AVX512 (comportant des unités capables de gérer des registres de largeur 512 bits) pour les processeurs de l'architecture Xeon Phi (MIC). Du côté des constructeurs concurrents, AMD a commencé par définir son propre jeu d'instructions baptisé 3DNow ! avant de décider d'utiliser les mêmes extensions qu'Intel depuis 2010. De leur côté, ARM et IBM proposent également leur propre extension du jeu d'instructions pour les calculs vectoriels. ARM a développé les instructions NEON tandis que IBM a conçu le jeu d'instructions Altivec. Même s'il existe des différences dans la définition et l'utilisation de ces différents jeux d'instructions, le principe reste le même que celui énoncé dans cette section.

À l'instar de la technologie multicœur et manycore, l'extension du jeu d'instructions vectorielles met l'accent sur l'utilisation de la pile logicielle pour améliorer les performances des applications. En effet, en théorie, doubler la taille des registres et unités vectoriels permet de doubler les performances calculatoires du processeur sous-jacent. En une seule instruction, à la place d'effectuer une opération par cycle (en régime permanent en considérant que les unités sont parfaitement pipelinées), il est possible de faire cette même opération sur plusieurs données pendant le même laps de temps. Le gain en performance crête est alors immédiat. Mais, pour profiter de cet accroissement de la puissance, il est nécessaire d'utiliser les instructions vectorielles dédiées. Ainsi, un code de calcul qui n'utilise pas, dans son code assembleur généré, ce type d'instruction ne pourra pas profiter du gain en performances. Il est alors nécessaire qu'un élément de la pile logicielle (ou le développeur d'application) fasse le nécessaire pour exploiter ces unités à travers les nouvelles instructions proposées. Même si cette exploitation est différente de celle des manycore, ces deux approches (élargissement des unités vectorielles et duplication des cœurs de calcul) sont deux pistes sérieuses et prometteuses pour proposer un gain de performance aux applications qui peuvent en profiter. La section suivante présente un processeur qui profite de ces deux mécanismes pour augmenter la puissance crête en calcul flottant.

1.2.5 Exemple du processeur Intel Xeon Phi

Comme nous l'avons vu dans les sections précédentes, il existe 3 axes pour améliorer les performances des processeurs modernes dédiés au calcul haute performance : optimisation du cœur, duplication des cœurs de calcul, élargissement des unités de calcul vectoriel. L'approche prise par Intel pour faire évoluer sa gamme de processeurs dédiés au monde du HPC essaie de jouer sur les trois tableaux en proposant un compromis moins radical que celui pris par NVIDIA avec l'ajout de composants de calcul flottant dans ses cartes graphiques. Ainsi, Intel a choisi de partir d'un cœur utilisé pour les architectures embarquées (par exemple les tablettes tactiles ou smartphones), d'ajouter quelques composants pour améliorer les performances flottantes, de dupliquer ce cœur plusieurs dizaines de fois et d'élargir d'un facteur deux les unités de calcul vectorielles passant ainsi à 512 bits. Il s'agit donc d'une solution qui propose un compromis entre les trois axes d'améliorations. Le tableau suivant résume les principales caractéristiques de ce processeur :

Nombre de cœurs physiques	60 (plus un cœur dédié au système)
Nombre de cœurs logiques	240 (4 hyperthreads/cœur)
Performances crêtes	1 TFlops double précision
Mémoire disponible	16GB
Bande passante mémoire	320GB/s
Registres vectoriels	512bits avec instruction Fused Multiply-Add (FMA) : $a \leftarrow a + b \times c$
Cache	32Ko pour le L1 et 32Mo au total pour le cache L2

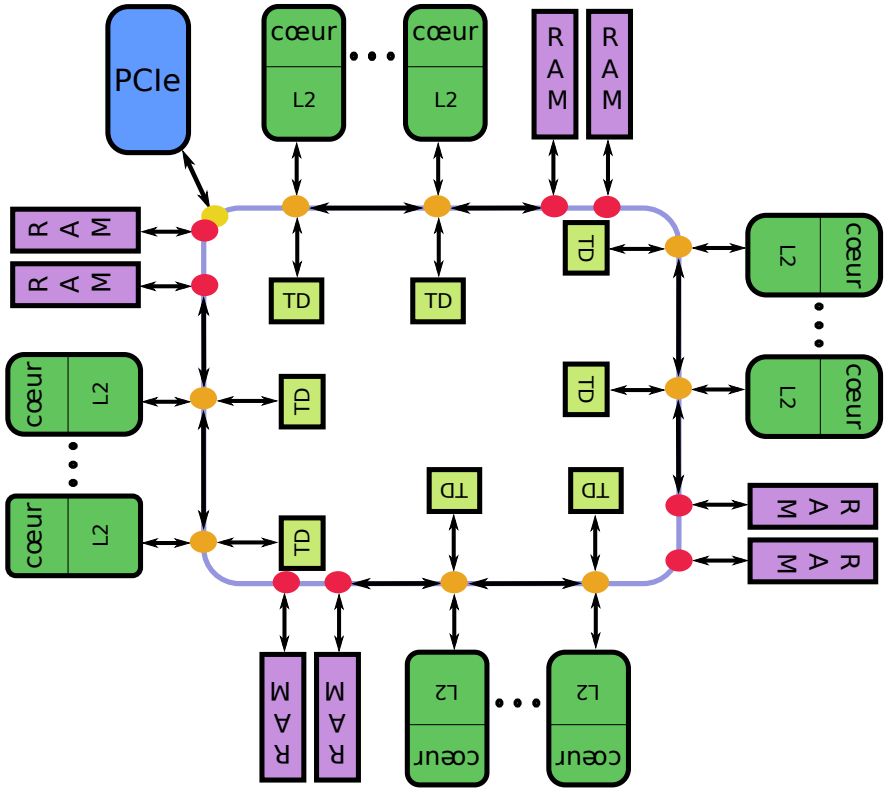


FIGURE 1.18 – Architecture d'un processeur Intel Xeon Phi (KNC)

La Figure 1.18 présente une vue logique de l'architecture globale du processeur Intel Xeon Phi. La principale originalité de ce processeur est l'organisation en anneau du cache de niveau 2 (L2) qui présente une partie privée à chaque cœur avec un cohérence qui s'effectue via cette structure en anneau et grâce au TD (Tag Directory) qui est également distribué [6]. Autour de cette structure, les cœurs sont disposés de manière symétrique entrelacée avec des canaux d'accès à la mémoire embarquée de la carte. La technologie de cette mémoire et le nombre de canaux permet d'atteindre une bande passante théorique de 320GB/s, ce qui est comparable à la bande passante disponible sur les GPGPUs NVIDIA de la même époque, basée de la technologie GDDR5.

À la genèse de cette puce massivement parallèle en 2009, Intel a d'abord annoncé et développé le Single-Chip Cloud Computer (SCC) qui dérivait du très expérimental « Teraflops Research Chip » en 2006 [8]. Le Xeon Phi, connu sous le nom de code KNight Corner (KNC), est ainsi la première version commerciale mise sur le marché de cette gamme. Elle a été annoncée officiellement à la conférence ISC 2012 [9] [46]. En effet, un premier prototype du nom de KNight Ferry (KNF) avait été développé quelques années auparavant, mais seuls quelques entreprises et instituts ont eu accès à ces machines. En revanche, il y a déjà d'autres évolutions prévues : le KNight Landing (KNL) dans un futur très proche [10] dont une photo de la puce est présentée dans la Figure 1.19 et après le KNight Hill (KNH) annoncé officiellement à la conférence Super Computing 2014.

6. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner
8. http://www.dailytech.com/Intel+Demonstrates+48Core+SingleChip+Cloud+Computer/article16951.htm
9. http://newsroom.intel.com/community/intel_newsroom/blog/2012/06/18/intel-names-the-technology-to-revolutionize-the-future-of-hpc--intel-xeon-phi-product-family
10. http://newsroom.intel.com/community/intel_newsroom/blog/2013/06/17/intel-powers-the-worlds-fastest-supercomputer-reveals-new-and-future-high-performance-computing-technologies

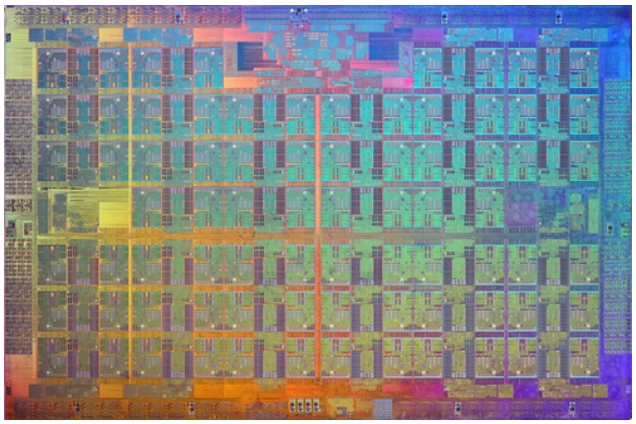


FIGURE 1.19 – Photo de la puce KNL [7]

1.3 Exploitation du parallélisme SIMD

Comme nous l'avons décrit dans la section précédente, le processeur Intel Xeon Phi est un exemple d'approche avec plusieurs niveaux de parallélisme disponible. En effet, on peut constater deux types distincts de parallélisme à exploiter pour atteindre le maximum de performances :

- Parallélisme inter-cœur : Ce processeur expose environ une soixantaine de cœurs qui peuvent être utilisés avec un ensemble de threads, de processus ou de processus multi-threadés.
- Parallélisme intra-cœur : Le Xeon Phi propose également un parallélisme intra-cœur sous plusieurs formes. Tout d'abord, la microarchitecture comporte un mécanisme d'exécution dans le désordre permettant de profiter du parallélisme d'instructions disponibles dans le code généré d'une application. En revanche, la faible taille de la fenêtre d'ordonnancement limite les possibilités de ce mécanisme. Pour pallier ce problème, le processeur propose plusieurs threads matériels par cœur (nommés hyperthreads) qui permettent de remplir les unités fonctionnelles avec des flots d'instructions indépendants. Le parallélisme d'instructions est exploité par des threads matériels (hyperthreads) vu que la puce est *in-order*. Pour rappel, les processeurs in-order exécutent les instructions comme suit :
  1. Les instructions sont récupérées ;
  2. Dans le cas où des opérandes d'entrées sont utilisés, l'instruction est envoyée vers son unité fonctionnelle ;
  3. L'instruction est exécutée ;
  4. L'unité retourne le résultat dans le registre destination.

Ainsi, on constate qu'il est nécessaire d'exploiter tous ces niveaux de parallélisme pour pouvoir tirer le plus de performances possible de ce type de processeur. Ces niveaux sont orthogonaux, même si des décisions d'exploitation d'un niveau peuvent influencer les performances obtenues par les autres niveaux. Plus précisément, une partie non négligeable des performances passent par l'exploitation des unités vectorielles grâce au jeu d'instructions SIMD. Cette technique se nomme vectorisation.

1.3.1 Historique de la vectorisation

La genèse du calcul vectoriel remonte à 1944 avec la machine Colossus 2 [59]. Mais, ce n'est qu'au début des années 60, avec l'arrivée des transistors que ce type d'unités fonctionnelles a commencé à se développer [71]. Pour exploiter ces nouvelles unités, la vectorisation automatique a émergé un peu plus tardive-

ment [2]. Il faudra attendre la sortie de la machine Cray 1 en 1975 pour voir apparaitre un supercalculateur purement vectoriel [19], même si les unités vectorielles ont été utilisées pour la première fois en calcul haute performance avec la machine CDC Star-100 [59]. Durant les décennies suivantes, ce type de calcul a été mis en arrière-plan dans les supercalculateurs, privilégiant la multiplication du nombre de cœurs de calcul et la complexité des processeurs. En effet, l'arrivée des architectures en grappe (« cluster ») [59] a tout d'abord été sans unités vectorielles intensives. Mais depuis quelques années elle tend à refaire surface afin de pouvoir permettre un meilleur passage à l'échelle des codes scientifique sur les prochaines générations de supercalculateurs.

1.3.2 Exemple de vectorisation manuelle

Afin de comprendre les enjeux et les difficultés de l'exploitation des unités vectorielles larges, nous allons nous intéresser à un code relativement réduit, mais comportant néanmoins un peu de flot de contrôle (plusieurs chemins possibles dans le programme) et de flot de données (accès indirect à la mémoire). Ce programme, nommé MC, est un code de simulation de transport de particules. Il repose sur une méthode Monte Carlo en discrétisant l'environnement sous la forme d'un maillage à une dimension (1D). Ce type d'approche peut être utilisée, par exemple, pour des calculs de criticité (centrale nucléaire) ou en imagerie médicale (radiographie). Cette mini application est relativement petite et donc est très simple à manipuler. Le problème étudié dans notre cas concerne une géométrie 1D qui représente un empilement de couches avec des niveaux d'absorption différents. Sur ce domaine, des particules apparaissent initialement en un seul point (source unique) au milieu et elles se propagent dans toutes les directions. Leurs histoires sont indépendantes, car elles n'interagissent pas entre elles. Nous suivons ainsi l'évolution de ces particules jusqu'à ce qu'elles quittent la géométrie. La grandeur calculée en sortie de ce code est l'énergie absorbée par chaque couche au passage des particules.

La fonction principale contient une boucle itérant sur le nombre de particules encore en vie. En effet, étant donné que le code commence avec un nombre fixe de particules situées dans la même cellule, la simulation se termine une fois que toutes ces particules ont quitté la géométrie. Ceci est implémenté avec une notion de nombre de particules en vie, qui tombe à 0 une fois la simulation terminée. À chaque itération de cette boucle principale, le code effectue les opérations suivantes :

1. Calcul de la distance entre les particules et les mailles voisines
2. Calcul de l'interaction avec les propriétés de la maille courante
3. Transfert de poids entre les particules et le maillage
4. Gestion du changement de maille d'une particule (si nécessaire)
5. Gestion de l'interaction avec le maillage (si nécessaire)

La première fonction appelée dans la boucle principale du programme MC est nommée `dist_sortie_couche`. Le code de cette fonction est présenté dans le Listing 1.1. Le but principal est alors de calculer la distance qui sépare chaque particule de la cellule voisine la plus proche. Ainsi, la fonction débute par une itération sur la totalité des particules (présente depuis le début de la simulation). Le premier test `if` porte ainsi sur le tableau `p_enable` permettant d'évaluer si la particule courante `ip` est encore active ou non. Dans le cas où la particule est déjà sortie du maillage, elle n'intervient plus dans la simulation. Il n'est alors pas nécessaire de continuer à la traiter. Dans le cas contraire, la distance par rapport à la face la plus proche (frontière entre deux mailles) est calculée grâce à la fonction `calc_coord_face`.

La deuxième fonction appelée dans la boucle principale de l'application MC est `dist_interaction` dont le pseudo-code est présenté dans le Listing 1.2. Elle permet de calculer les distances aux interactions avec le milieu contenu dans les mailles. Ici, la notion de distance relève d'un tirage aléatoire en fonction des

Listing 1.1 – Fonction dist_sortie_couche initiale

```
void dist_sortie_couche(...){
    for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
        if (p_enable[ip]){
            const real_t mu = p_mu[ip];
            real_t di = MAXREAL;
            int ev = 0;
            if (mu < -EPS_PRECIS || EPS_PRECIS < mu){
                const int i_f = p_nc[ip] + (mu < 0 ? 0 : 1);// i_f : indice de face de sortie
                const real_t xf = calc_coord_face(domaine, i_f);
                di = (xf - p_x[ip]) / mu;
                ev = i_f;
            }
            p_di[ip] = di;
            p_ev[ip] = ev;
        }
    }
}
```

Listing 1.2 – Fonction dist_interaction initiale

```
void dist_interaction(...){
    const real_t i_rate = domaine->interaction_rate;
    for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
        if (p_enable[ip]){
            const real_t h = rnd_real(p_sd[ip]); // h app. a [0, 1]
            const int ic = p_nc[ip]; // Indice de la couche
            const real_t sig_i = i_rate*c_sig[ic];
            real_t di = MAXREAL;
            if (sig_i > EPS_PRECIS){
                di = -log(h) / sig_i;
            }
            if (di < p_di[ip]) {
                p_di[ip] = di;
                p_ev[ip] = -1;// Alors l'événement est interaction
            }
        }
    }
}
```

caractéristiques du maillage et, plus particulièrement, de la maille dans laquelle est située la particule. La structure de cette fonction est identique à la précédente : une boucle parcourt toutes les particules et ne traite que celles qui sont encore situées dans le maillage, c'est-à-dire en vie. En fonction de l'évaluation de cette distance (tableau `p_di`), le tableau permettant d'évaluer l'événement qui prime est mis à jour. Ainsi, le tableau `p_ev` est écrit avec la valeur `-1` si le prochain événement à traiter est lié à l'interaction avec le milieu ou une valeur positive sinon (correspondant à la maille voisine de sortie).

Une fois les distances aux différents événements évaluées, la particule contribue à la maille sur laquelle elle se trouve à travers un transfert de poids. La fonction `absorption` implémente ce traitement à travers le pseudo-code présenté dans le Listing 1.3. Encore une fois, la structure de ce bout de code est très similaire aux fonctions précédemment vues : une boucle itère sur la totalité des particules traitées depuis le début de la simulation. Si la particule courante `ip` est encore à traiter (donc dans le maillage), le poids de transfert est calculé dans la variable `dw` avant d'être retiré à la particule (tableau `p_wmc`) pour être ajouté à la maille courante (tableau `c_wa`).

Une fois que la particule a contribué à sa maille, il faut gérer les événements en commençant par la sortie de couche. Toujours avec un code se rapprochant des autres, la fonction `sortie_couche`, dont le pseudo-code

Listing 1.3 – Fonction absorption initiale

```
void absorption(...){
    const real_t a_rate = domaine->absorption_rate;
    for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
        if (p_enable[ip]){
            const int ic = p_nc[ip]; // Indice de la couche
            const real_t sig_a = a_rate*c_sig[ic];
            const real_t wmc = p_wmc[ip];
            const real_t di  = p_di[ip];
            const real_t dw = (1 - exp(-sig_a*di)) * wmc;

            // Le poids retiré de la particule est déposé
            // dans la couche dans laquelle la particule se déplace
            p_wmc[ip] -= dw;
            c_wa[ic]  += dw;
        }
    }
}
```

Listing 1.4 – Fonction sortie couche initiale

```
void sortie_couche(...){
    *nb_disable = 0;
    for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
        if (p_enable[ip]){
            if (p_ev[ip] ≥ 0 /* condition sortie couche */){
                const int i_f = p_ev[ip];// Indice de face de sortie de la couche

                // On positionne la particule EXACTEMENT sur la face
                p_x[ip] = calc_coord_face(domaine, i_f);

                // Determination de l'indice de la nouvelle couche
                p_enable[ip] = !(i_f == 0 || i_f == domaine->nb_couches);

                if (p_enable[ip]){
                    const int old_nc = p_nc[ip];
                    p_nc[ip] = (i_f == old_nc ? old_nc-1 : old_nc+1);
                }
                *nb_disable += int(!p_enable[ip]);
            }
        }
    }
}
```

est détaillé sur le Listing 1.4, commence par itérer sur toutes les particules `ip`. Ensuite si `ip` est active et que son événement `p_ev[ip]` est « sortie couche » (≥ 0), sa nouvelle position est calculée et stockée dans `p_x`. Ce qui permet de faire le changement de couche stocké dans `p_nc[ip]` et de désactiver la particule si les bords du domaine sont atteints en mettant à jour `p_enable[ip]`.

L'autre événement est l'interaction. Toujours avec un code similaire, bien que plus simple, celui-ci est visible sur le Listing 1.5. Cette fonction commence comme les précédentes à itérer sur les particules `ip` puis à s'assurer que la particule est en toujours en vie en vérifiant dans le tableau `p_enable`. Après nous avons le test pour vérifier si l'événement est bien celui d'interaction en s'assurant que `p_ev[ip]` vaut -1. La fonction ensuite se compose de deux parties : stockage dans `p_x` de la nouvelle coordonnée. Enfin la nouvelle direction de la particule est tirée au sort et mis dans `p_mu`.

Ce code peut paraitre très simple, mais il illustre quelques difficultés pour exploiter les unités vectorielles. Ainsi, nous avons exploré trois possibilités pour apporter de la vectorisation sur ce programme : (i) vectori-

Listing 1.5 – Fonction interaction initiale

```
1  void interaction(...){
2      for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
3          if (p_enable[ip]){
4              if (p_ev[ip] == -1 /* code signifiant interaction */){
5                  p_x[ip] += p_mu[ip] * p_di[ip];// Deplacement de la particule dans la couche
6                  p_mu[ip] = 2*rnd_real(p_sd[ip]) - 1;// Tirage d'une nouvelle direction
7              }
8          }
9      }
10 }
```

Listing 1.6 – Vectorisation guidée de la fonction interaction

```
1  void interaction(...){
2      #pragma simd
3      for(int ip = 0 ; ip < ens_partic->nb_partics ; ip++){
4          if (p_enable[ip]){
5              if (p_ev[ip] == -1 /* code signifiant interaction */){
6                  p_x[ip] += p_mu[ip] * p_di[ip];// Deplacement de la particule dans la couche
7                  p_mu[ip] = 2*rnd_real(p_sd[ip]) - 1;// Tirage d'une nouvelle direction
8              }
9          }
10     }
11 }
```

sation automatique, (ii) vectorisation guidée et (iii) vectorisation manuelle.

Vectorisation automatique

La première approche possible pour exploiter les unités vectorielles larges dans la mini application MC est de laisser le compilateur, par exemple celui d'Intel, essayer lui même de vectoriser les différentes boucles. Dans cette approche, seul le compilateur est responsable pour étudier le potentiel d'utilisation de telles unités dans le code. Il se concentre alors sur les boucles de calcul qui, a priori, possède un potentiel plus grand qu'une séquence continue d'instructions. En effet, l'exploitation des ressources vectorielles passe par l'expression d'un parallélisme de données en émettant des instructions qui effectuent le même type de calcul sur des données différentes. Il est alors pertinent de penser qu'un tel parallélisme peut être disponible entre plusieurs itérations d'une même boucle.

Une fois une boucle ciblée, elle doit remplir un certain nombre de critères pour être transformée et exploiter le parallélisme vectoriel. Ainsi, son domaine d'itération doit être préférablement fini (boucle `for`) et les itérations doivent être indépendantes. Cette dernière contrainte est une des clés de la vectorisation. En effet, le compilateur ne peut pas transformer la boucle et exécuter conjointement plusieurs itérations consécutives s'il existe une dépendance entre deux itérations consécutives. Le compiler procède ainsi à une passe d'analyse de dépendances de données pour statuer si la transformation en vue de la vectorisation est légale ou non.

Vectorisation guidée

Dans le cas où une des premières phases ne se déroulerait pas correctement dans le compilateur, il est possible de guider le processus de vectorisation. En effet, nous avons vu dans la section précédente que la vectorisation automatique passe par plusieurs étapes, dont l'analyse de dépendances et la profitabilité de la

Listing 1.7 – code d'interaction avec des intrinsics

```
1  void interaction(...){
2      int n=ens_partic→nb_partics;
3      int  i = 0, tail, nn;
4      int is=-1;
5      for (int i = 0; i < n; i+=16){
6          is++;
7          __mmask16 mask2=
8              _mm512_mask_cmpeq_epi32_mask(masks_p_enable[is],p_ev[is],_mm512_set1_epi32(-1));
9          if(_mm_countbits_32(masks_p_enable[is])>0){
10             __m512 tmpmul = _mm512_mask_mul_ps(p_x[is],mask2,p_di[is],p_mu[is]);
11             p_x[is]= _mm512_mask_add_ps(p_x[is],mask2,p_x[is],tmpmul);
12
13             real_t pmu[16];
14             _mm512_packstorelo_ps((void*)pmu,p_mu[is]);
15             _mm512_packstorehi_ps(((char*)(pmu))+64,p_mu[is]);
16
17             for(int k=0 ; k<16 ; k++){
18                 if((mask2&(1<<(k)))!=0){
19                     pmu[k]=2*rnd_real(p_sd[i+k])-1;
20                 }
21             }
22
23             p_mu[is]=_mm512_loadunpacklo_ps(p_mu[is],(void*)(&pmu[0]) );
24             p_mu[is]=_mm512_loadunpackhi_ps(p_mu[is],((char*)(pmu))+64 );
25         }
26     }
27 }
```

transformation. Ces deux étapes peuvent être neutralisées en utilisant une directive de compilation notée `#pragma simd` en C/C++

Le Listing 1.6 présente le pseudo-code de la fonction `interaction` de notre petite application MC après avoir appliqué la vectorisation guidée. On peut noter que, d'une part, la modification du code est minime, uniquement la ligne 2 est ajoutée et que, d'autre part, cette transformation est indépendante de l'architecture cible. Mais ce coup de pouce n'influence que les premières étapes de la vectorisation. En indiquant au compilateur que la boucle ne porte pas de dépendances de données et en biaisant le modèle de coût de profitabilité, le code généré peut être sous-optimal en fonction de la complexité du corps de la boucle. La partie suivante décrit une autre méthode pour pallier ces problèmes.

Vectorisation manuelle

Dans le cas où la vectorisation guidée n'est pas suffisante (par exemple si les techniques de transformations et génération de code du compilateur ne sont pas assez efficaces vis-à-vis des capacités de l'architecture cible), il est alors possible d'utiliser l'approche de vectorisation manuelle pour comprendre les limites des compilateurs. Cette approche consiste alors à exprimer le code sous forme vectorielle avec une taille de vecteur fixe pour limiter le travail du compilateur. Pour ce faire, Intel fournit des fonctions intrinsèques (intrinsics) utilisables en C/C++ qui correspondent plus ou moins directement à des instructions assembleur. Le compilateur n'a pas alors à faire une analyse de dépendances de données, ni un calcul de profitabilité, ni les transformations de boucles nécessaires pour exploiter le parallélisme vectoriel. Il suffit de finaliser la génération de code en utilisant les instructions correspondantes et en effectuant quelques optimisations et allocations de registres.

Dans le Listing 1.7, la fonction interaction a été vectorisée manuellement. Nous pouvons noter que le code

transformé est devenu beaucoup plus verbeux. De plus, cet exemple montre la suppression du contrôle de flot (le test `if` sur la particule courante) pour le remplacer par un masque. Un masque est un vecteur de booléen permettant aux instructions vectorielles d'effectuer une opération uniquement sur certains éléments d'un vecteur [59].

Résultats

TABLE 1.1 – Accélération de MC en fonction de la version

Version	Accélération
Vectorisation automatique	1
Vectorisation guidée	2.52
Vectorisation manuelle	4.91

Sur le Tableau 1.1 nous pouvons observer l'accélération des deux versions optimisées relative au temps de la vectorisation automatique, qui correspond au code d'origine compilé sans aucune option particulière. Ces résultats ont été faits sur l'architecture KNC avec ICC 14. La première version optimisée est celle « guidée » c'est-à-dire avec les pragma SIMD sur les boucles de calculs, son accélération est non négligeable, mais améliorable : 2.52 sur un maximum de 8. La seconde version est celle « manuelle » dans laquelle l'accélération est meilleure, mais encore perfectible : 4.91 sur 8, même si c'est mieux que l'autre cela a demandé beaucoup plus de travail et n'est pas applicable sur un code scientifique conséquent.

1.4 Synthèse

Nous avons vu que la conception d'ordinateurs dédiés pour le calcul remonte aux années 1950, mais c'est en 1964 que le premier supercalculateur a été fabriqué. Néanmoins pour arriver aux machines telles qu'elles sont aujourd'hui il a fallu attendre 1980 pour la conception d'un premier cluster, agrégats de machines standards. Toutes ces évolutions ont résulté en l'accroissement de la puissance de calcul. Celui-ci s'est fait en plusieurs étapes : (i) le passage de la triode au transistor qui a permis l'émergence des CPU, (ii) l'augmentation de la fréquence de ces puces, (iii) l'augmentation de leur nombre de cœurs et (iv) l'élargissement des unités de calcul. Cette dernière étape a amené avec elle le parallélisme SIMD et la notion de vectorisation. Celle-ci peut-être apportée de diverses manières dans un code C/C++ : (i) en laissant le compilateur s'en occuper tout seul, (ii) en rajoutant des annotations sur les boucles à vectoriser, le `#pragma SIMD` et (iii) en modifiant la structure du code pour rajouter des intrinsics. Sur un petit code nous avons pu étudier l'impact de ces techniques d'optimisation et les résultats nous ont montré que la manière (iii) s'avère plus performante que la (ii), mais aussi que la (iii) n'est pas applicable simplement. Donc nous sommes à la recherche d'une technique qui modifie le code pour qu'il soit plus favorable à la vectorisation et donc pour rendre plus efficace l'ajout des pragma.

Chapitre 2

État de l'art sur la vectorisation de code irrégulier

Dans le chapitre précédent, nous avons présenté une partie de l'évolution des architectures de processeurs pour le calcul haute performance. Ceci nous a permis d'identifier un composant qui a pris de l'importance durant ces dernières années : les unités de calcul vectorielles. En effet, l'augmentation de la taille des registres SIMD permet d'améliorer les performances des processeurs d'un facteur non négligeable. Par exemple, une utilisation efficace de ces unités peut représenter 7/8 des performances totales de calcul d'un processeur Intel Xeon Phi. Mais nous avons également mis en avant leur utilisation complexe à travers un exemple simple (nommé MC). Avec un flot de contrôle et de données non linéaire, il devient plus compliqué de tirer des performances des registres SIMD. En revanche, en transformant le code à la main, il semble tout à fait possible d'espérer un gain important en exploitant finement les instructions à notre disposition. Ainsi, ce chapitre propose un état de l'art des techniques d'exploitation des unités vectorielles sur processeur récent pour exploiter le parallélisme de données au niveau SIMD d'une application. Nous présenterons également les techniques qui peuvent s'appliquer aux codes dits irréguliers aussi bien au niveau du flot de contrôle que du flot de données.

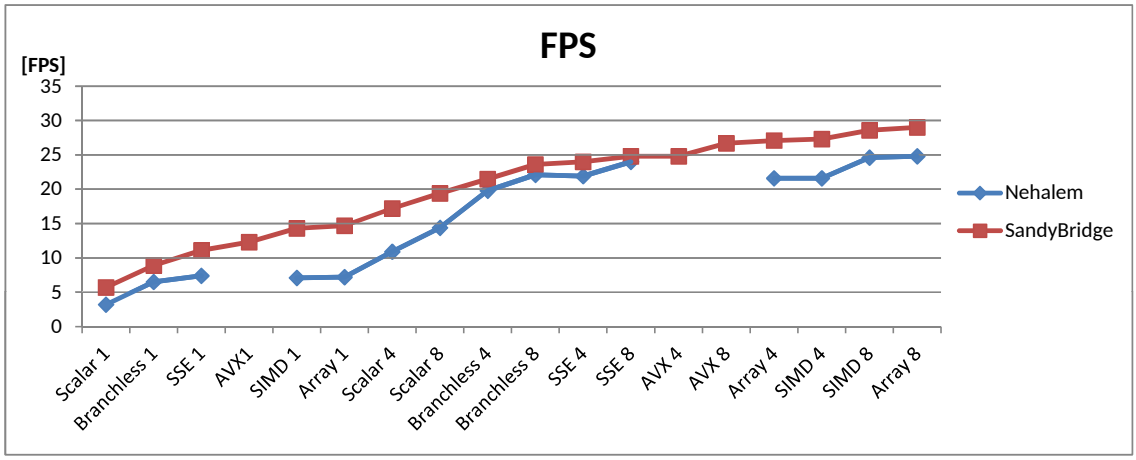


FIGURE 2.1 – Résultat obtenu par KRISTOF et al.[40]

Pour illustrer un peu plus la difficulté de l'approche, la Figure 2.1 présente les performances obtenues sur un benchmark simulant le moteur de rendu d'un jeu vidéo nommé Intel Smoke Engine [40]. La contrainte de

devoir fournir un rendu à plus de 30 images par seconde met la pression sur les performances pour améliorer le confort de jeu. C'est pourquoi cette application a été parallélisée sur des threads et a été optimisée en vue d'utiliser les capacités de calculs vectoriels des processeurs Intel Nehalem et Intel Sandy Bridge. Le benchmark de la Figure 2.1 représente ainsi le nombre d'images par seconde en fonction de différentes versions du moteur. Il consiste à calculer le rendu des 2500 premières images en ignorant le temps passé à charger la scène prédéfinie du disque dur. Les différentes versions en abscisses sont suffixées par le nombre de threads utilisés dans la simulation pour l'exécution sur un processeur multicoeur. Le reste du nom décrit une version du code :

— Scalar : le code d'origine
— Branchless : remplacement du code d'origine implémentant les tests de collision avec une version scalaire sans branchements
— SSE : remplacement du code d'origine implémentant les tests de collision avec une version contenant des appels intrinsèques SSE
— AVX : remplacement du code d'origine implémentant les tests de collision avec une version contenant des appels intrinsèques AVX
— SIMD : ajout de directives SIMD au code d'origine
— Array : modification du code d'origine avec l'utilisation de notation sous forme de tableau (array notation)

Cette comparaison montre qu'il est possible d'obtenir un gain significatif en adaptant le code pour exploiter les unités vectorielles. Elle illustre ainsi plusieurs solutions : la vectorisation automatique (guidée ou non) ainsi que la vectorisation explicite.

Ce chapitre traite des travaux existants selon ces deux axes. Tout d'abord, la section 2.1 détaille l'approche de la vectorisation explicite en s'appuyant sur une extension de langage ou sur une bibliothèque pour exploiter les unités vectorielles. Ensuite, la section 2.2 traite des travaux existants sur la vectorisation automatique, permettant d'extraire un parallélisme vectoriel à partir d'un code séquentiel composé de boucles ou de fonctions. Enfin, nous exposons les transformations de code utiles pour permettre d'aider la vectorisation automatique dans la section 2.3 avant de conclure sur l'approche que nous allons suivre dans le reste du document.

2.1 Vectorisation explicite

Dans l'introduction de ce chapitre, nous avons vu un exemple d'application qui pouvait bénéficier d'une écriture particulière appelée *array notation*. Dans ce domaine nommé vectorisation explicite, deux approches existent qui nécessitent de revoir complètement la manière de programmer en changeant, par exemple, les structures de données.

La première approche se base sur un langage existant en l'étendant avec de nouveaux mots clefs pour faciliter l'exploitation du parallélisme vectoriel. Par exemple, Sierra est une bibliothèque qui étend le langage C++ pour rajouter la notion de vecteur via un nouveau mot clef nommé *varying* [43] acceptant un paramètre qui représente la taille du vecteur. Ainsi, la taille des données manipulées par le programme est précisée par l'utilisateur et le compilateur pourra utiliser cette notion de tableaux pour exploiter les registres vectoriels et les unités de calcul associées. Les Listings 2.1a et 2.1b présentent un exemple d'utilisation de ce langage étendu. Le premier bout de code illustre une fonction scalaire nommée `raymarch` écrite en C++ classique. La version Sierra est présentée au Listing 2.1b. Les déclarations scalaires ont été étendues en déclarations vectorielles grâce à l'ajout du mot clé `varying` avec une taille de vecteur `L`. Avec ces modifications, le compilateur peut générer un code vectoriel lors des opérations entre variables possédant le même attribut.

```
float
raymarch(float volume[], Ray& ray,/*...*/) {
    float rayT0 , rayT1;
    if(!intersect(ray,bounding_box,
       rayT0 , rayT1))
           return 0.f;
    float result = 0.f;
    auto pos = ray.dir*rayT0 + ray.origin;
    auto t = rayT0;
    while (t < rayT1) {
        auto d=density(pos, volume ,/*...*/);
        auto atten = /*...*/;
        if(atten  > THRESHOLD)
            break;
        auto light=compute_lighting(/*...*/);
        result  +=  light */*...*/;
        pos +=/*...*/;
        t += /*...*/;
    }
    return gamma_correction(result);
}
```

a – code séquentiel[43]

```
float varying(L)
raymarch(float volume[],
         Ray varying(L)& ray,/*...*/){
    float  varying(L) rayT0 , rayT1;
    if (!intersect(ray, bounding_box,
        rayT0 , rayT1))
            return 0.f;
    float varying(L) result = 0.f;
    auto pos = ray.dir*rayT0 + ray.origin;
    auto t = rayT0;
    while (t < rayT1) {
        auto d=density(pos, volume ,/*...*/);
        auto atten = /*...*/;
        if(atten  > THRESHOLD)
            break;
        auto light=compute_lighting(/*...*/);
        result  +=  light */*...*/;
        pos +=/*...*/;
        t +=/*...*/;
    }
    return gamma_correction(result);
}
```

b – code vectoriel [43]

Listing 2.2 – Exemple de code utilisant Sierra [43]

Listing 2.1 – Exemple de code avec boost : :simd[24]

```
#include <boost/simd/pack.hpp>
using namespace boost::simd ;
int main ( )
{
    float s , tx[] = {1,2,3,4} ;
    // Build pack from memory and values
    pack<float> x(tx , tx+4) ;
    pack<float> a(1.37) , b(1,-2,3,-4),r ;
    // Operator and functions-calls
    r += min (a*x+b , b) ;
    // Array interface
    r[0] = 1. f + r[0];
    // Range interface : using std::accumulate
    s = accumulate(r.begin(), r.end(), 0.f) ;
    return 0;
}
```

La deuxième approche possible pour exploiter les unités vectorielles grâce à une technique de vectorisation explicite se base sur l'utilisation de bibliothèques implémentant des traitements vectoriels. Ainsi, Boost SIMD [24] est une bibliothèque pour abstraire la vectorisation en C++ via des templates, proposée en tant que potentielle extension de la bibliothèque Boost [1]. Les différentes fonctions des templates vont être traduites en fonctions bas niveaux qui travaillent sur les registres en tenant compte de l'architecture. Il y a donc besoin de réécrire le code à la main (changement des déclarations, utilisation de fonctions spécifiques, ...) si l'on veut rajouter de la vectorisation à un code existant. Par exemple, le Listing 2.1 présente un code très simple qui repose sur la bibliothèque BOOST pour exploiter le parallélisme vectoriel. Ainsi, en plus de l'inclusion un fichier en-tête spécifique, les déclarations scalaires sont changées en déclarations templatisées vectorielles (paquet de réels en simple précision). Ensuite, les fonctions fournies par le langage C++ (comme `accumulate`) existent aussi en version qui travaille sur des vecteurs. Enfin, une autre limite qui peut être

1. http://www.boost.org/

Listing 2.2 – Exemple simple de vectorisation [11]

```
1 char a[N], b[N], c[N];
2 ...
3 for (i = 0; i < N; i++) {
4   a[i] = b[i] + c[i];
5 }
6
```

```
Back: movq mm0, _b[ecx] ; load 8 bytes from b
      paddb mm0, _c[ecx] ; add 8 bytes from c
      movq _a[ecx], mm0 ; store 8 bytes into a
      add ecx, 8 ;
      cmp ecx, edi ;
      jl Back ; looping logic
```

gênante en HPC est l'obligation d'utiliser le langage C++.

2.2 Vectorisation automatique de boucles

Une autre approche consiste à laisser faire le compilateur pour générer du code vectoriel à partir d'un code écrit de façon scalaire. Cette technique s'appelle la vectorisation automatique et s'applique principalement sur des boucles de calcul qui exposent du travail répétitif et similaire sur des données différentes, parfois contiguës. Cette section décrit ainsi les techniques d'exploitation du parallélisme vectoriel au sein des boucles de calcul. Nous énonçons tout d'abord le principe général en discutant sur les transformations possibles d'une boucle simple. Ensuite, nous exposons les travaux existants traitant des boucles avec un flot de contrôle complexe ainsi que les nids de boucles (parfaits ou non).

2.2.1 Parallélisme vectoriel d'un niveau de boucle

Intéressons-nous tout d'abord à la gestion d'un seul niveau de boucle. Dans le chapitre précédent, nous avons déjà évoqué la notion d'extraction de parallélisme vectoriel à travers la transformation manuelle du code MC. Il était alors question de manipuler les boucles qui parcourent les particules de telle sorte que les unités de calcul vectoriel puissent traiter plusieurs paquets de particules. Mais cette génération de code n'est pas tout le temps légale. Ainsi, la première étape pour exploiter le parallélisme vectoriel à partir d'une boucle est de vérifier qu'il est légitime d'exécuter plusieurs itérations de façon concurrente et synchrone. En effet, au lieu d'exécuter les itérations dans l'ordre, plusieurs itérations seront exécutées de façon simultanée en faisant progresser les calculs étape par étape. Une condition suffisante pour pouvoir effectuer ce genre de transformations est de garantir que les itérations de la boucle cible sont indépendantes.

Pour illustrer le principe de base de la vectorisation automatique, prenons l'exemple du Listing 2.2 issu des travaux de BIK et al. [11]. La partie gauche de cette figure expose une boucle très simple comprenant un seul niveau de profondeur (itérateur `i`). Le corps de cette boucle se compose uniquement d'une seule instruction C qui additionne des éléments de deux vecteurs pour stocker le résultat dans un troisième. Comme décrit précédemment, la première étape importante pour la légitimité de la vectorisation concerne les dépendances entre les itérations. Dans notre cas, en se focalisant uniquement sur le code de la boucle, il est possible qu'il existe une dépendance. En effet, si les tableaux `a`, `b` et `c` se chevauchent en mémoire, la lecture d'un élément de `b` peut entrainer un accès à la même case mémoire que l'écriture d'un élément de `a` à une itération qui précède. Dans ce cas, il existerait alors une dépendance de flot (lecture après écriture, ou RAW) qui pourrait empêcher l'exécution simultanée de plusieurs itérations consécutives. Mais cette situation n'apparait pas dans notre bout de code, car la première ligne illustre la déclaration de nos vecteurs. Chaque tableau comporte `N` éléments qui sont distincts en mémoire.

La vectorisation étant légitime, la partie droite du Listing 2.2 propose le code assembleur vectoriel x86 correspondant au résultat. En admettant que le registre `ecx` contienne la valeur de l'itérateur `i`, la première

ligne charge les 8 prochains éléments du tableau `b` (opération `movq`) tandis que la deuxième instruction charge également les 8 prochains éléments de `c` et additionne ces deux valeurs. Cette addition est effectuée par l'opération `paddb` qui signifie que les registres doivent être considérés comme des vecteurs d'octets. Ainsi, 8 opérations d'addition sont faites en une seule instruction assembleur. La suite du code est très simple : la troisième instruction stocke le résultat de cette addition dans le tableau `a`, finissant ainsi les 8 prochaines itérations. L'itérateur est alors incrémenté de 8 avec l'instruction qui suit, avant de reboucler si nécessaire.

Il est alors intéressant de se poser la question suivante : comment est-on passé du code en C au code assembleur sur l'exemple du Listing 2.2 ? Une première technique consiste à appliquer du déroulage de boucle (*Loop Unrolling*) avec un facteur cible K ($K = 8$ dans notre exemple précédent). On obtient alors K copies du corps de boucle, avec un itérateur qui est incrémenté de K à chaque itération (après avoir appliqué quelques transformations pour propager les différents incréments intermédiaires de l'itérateur en question). Une fois cette opération effectuée, il suffit alors de déplacer les opérations similaires pour les transformer en opérations vectorielles (c.-à-d., même opérateur, données différentes). Cette technique a été étendue et formalisée sous le nom de *Superword-Level Parallelism* (SLP [41]). En effet, le concept de super-mot désigne une instruction qui effectue une opération sur un ensemble de données au lieu d'un scalaire unique. Le principe du déroulage de boucles et le regroupement des opérations similaires permettent alors de créer des super-mots, facilitant ensuite le reste des optimisations du compilateur et le choix des instructions assembleur lors de la génération de code.

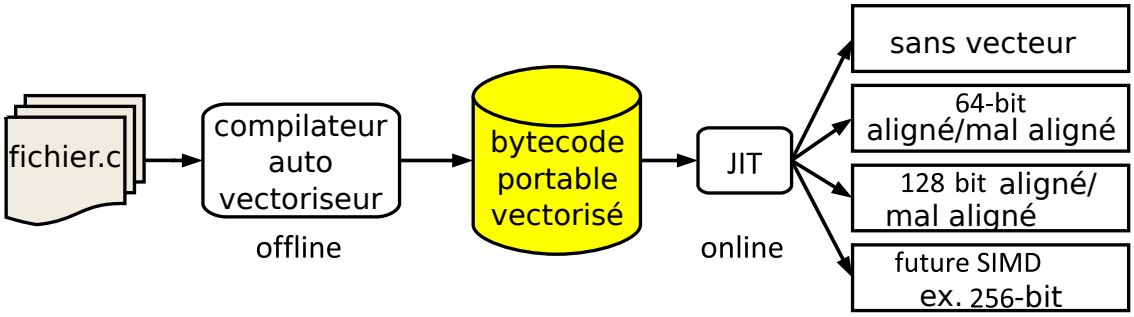


FIGURE 2.2 – Fonctionnement de Vapor SIMD par NUZMAN et al.[51]

Mais ce principe reste très général et plusieurs problèmes peuvent apparaitre, même sur une boucle très simple (contenant un seul bloc de base). En effet, en fonction des contraintes de l'architecture et du jeu d'instructions associé, certaines opérations peuvent demander un environnement précis pour une exécution optimale. Ainsi, un des premiers paramètres dont les compilateurs doivent tenir compte est la largeur des unités vectorielles. En effet, d'une génération d'architecture à une autre, le nombre de registres et leur taille peuvent varier du simple au quadruple, comme nous l'avons vu dans le chapitre précédent lors de la description sur histoire de l'apparition de ce type de registres. Enfin, un autre paramètre important est l'alignement des données sur une frontière fixe. En effet, au lieu de charger les données scalaire par scalaire, une boucle vectorisée aura tendance à émettre des opérations de chargement mémoire par paquets contigus. Dans ce cas, il existe deux possibilités : soit ce chargement est aligné sur une frontière (par exemple une ligne de cache), soit il ne l'est pas. Dans la première situation, chaque chargement donnera lieu au transfert d'une ligne de cache, ce qui sera, a priori, le meilleur scénario. Dans le second cas, il est alors nécessaire de charger de multiples lignes de cache pour former le vecteur souhaité. Chaque architecture peut réagir de façon différente : soit en gérant ce cas, quitte à pénaliser les performances, soit en refusant d'exécuter ce genre de chargement. Il devient alors nécessaire de s'adapter au comportement dynamique de l'architecture. Pour pallier ce problème, certaines approches dont Vapor SIMD [51], proposent une spécialisation dynamique du code vectoriel. La Figure 2.2 illustre la chaîne d'utilisation de cette plateforme : le code source (nommé

Listing 2.3 – Exemple de vectorisation intra-registre[11]

```
1 int a[N];                    Back: movq     mm1, _a[eax*4] ; charge 2 dwords depuis a
2 ...                                movq     mm0, mm2       ; copie 2 dwords avec 10
3 for (i = 0; i < N; i++) {          pcmpgtd  mm0, mm1       ; GT (supérieur à)  2 dwords
4   if (a[i] < 10)                   pandn    mm0, mm3       ; masque 2 dwords avec 1
5     a[i] = 0;                      movq     _a[eax*4], mm0 ; enregistre 2 dwords dans a
6   else                             add      ecx, 2         ;
7     a[i] = 1;                      cmp      ecx, eax       ;
8 }                                  jl       Back           ; saut
```

`fichier.c`) est compilé avec un compilateur optimisant se reposant sur des techniques de vectorisation. La finalité du code généré n'est pas de l'assembleur, mais une forme intermédiaire nommée *bytecode*. Ce dernier est portable selon les différents paramètres que nous venons d'évoquer. La véritable génération de code vers des instructions assembleur a lieu lors de l'exécution du programme grâce à un compilateur au vol (*Just in Time* ou JIT) qui pourra alors connaitre les caractéristiques réelles de l'architecture cible et du placement des données. Cette figure montre ainsi la génération de plusieurs variantes pour une même boucle de calcul.

Support d'un seul niveau de flot de contrôle

Mais les différents paramètres évoqués dans la section précédente ne sont pas les seuls obstacles à la vectorisation automatique. En effet, la complexité du flot de contrôle peut également entrainer des complications dans les techniques présentées. Pour illustrer ce problème, le Listing 2.3 propose, dans sa partie gauche, un exemple simple avec une boucle qui contient un bloc `if` et un bloc `else` associé. Contrairement à tous les exemples décrits précédemment dans ce chapitre, celui-ci possède un flot de contrôle qui peut varier d'une itération à l'autre. En effet, en fonction des valeurs du tableau `a`, la branche `if` peut être prise ou la branche `else`. Il est important de noter que, dans beaucoup de cas de figure, il est très difficile de déduire statiquement les chemins que vont suivre les itérations de ce genre de boucle. Cette remarque est la clé pour comprendre comment le flot de contrôle peut influencer l'exécution vectorielle. En effet, l'utilisation des unités vectorielles requiert l'expression d'une opération sur des données différentes et contiguës. L'instruction à la ligne 5 de l'exemple Listing 2.3 est un candidat simple pour l'utilisation de ces unités (instruction d'affectation de 0 dans plusieurs cases mémoire). Mais, à cause de la condition ligne 4, il est possible que l'instruction ligne 5 ne soit pas exécutée au profit de la ligne 7. Il est alors important de ne pas écrire 0 dans `a[i]`.

Il existe plusieurs approches, malgré tout, pour vectoriser une boucle qui contient un tel flot de contrôle. La première consiste à utiliser des masques afin d'affecter des valeurs différentes en fonction des itérations. Cette technique est très spécifique et peut s'appliquer uniquement dans les cas où les chemins ne diffèrent que par des valeurs (et non pas des instructions). L'exemple Listing 2.3 correspond à cette situation, il est alors possible de créer un vecteur contenant la valeur 1 quand les éléments sont supérieurs ou égaux à 10 (registre `mm0`). Il suffit ainsi de faire un `ET` binaire avec un registre qui ne contient que des valeurs 1 pour simuler l'affectation des valeurs 0 ou 1 dans le tableau `a`.

Listing 2.4 – Exemple Open Computing Language (OpenCL) [63]

```
1  __kernel void func(
2  __global int *A,
3  __global int *B,
4  __global int *C)
5  {
6      size_t idx = get_global_id(0);
7      if (idx > 6)
8          A[idx] = B[idx]* D[idx];
9      else
10         C[idx] = C[idx] * idx;
11 }
```

Listing 2.5 – Transformation après if-conversion (prédicat) [63]

```
1  P1 = (idx > 6)
2  P2 =  ~P1
3  [P1] b = load B[idx]
4  [P1] d = load D[idx]
5  [P1] t = mul b, d
6  [P1] store t → A[idx]
7  [P2] c = load C[idx]
8  [P2] t = mul c, idx
9  [P2] store t → C[idx]
```

Listing 2.6 – code après if conversion avec select et masque [66]

```
1  mask = (idx > 6)
2  b = B[idx] * D[idx]
3  c = C[idx] * idx
4  t = select(mask,b, c)
```

Une approche plus générique se basant sur ce même aspect optimiste s'appelle la *if-conversion* [63]. Elle consiste basiquement à transformer toutes les dépendances de contrôle (c'est-à-dire, branchements) en dépendances de données. Un exemple est présenté Listing 2.4 et Listing 2.5. Le code d'origine est présenté avec le langage de programmation OpenCL qui fonctionne sous forme de noyau de calcul en décrivant le travail que fait chaque thread. Ainsi, la fonction `func` correspond au point d'entrée un tel noyau grâce au mot clé `__kernel`. Cela signifie que cette fonction sera appelée par un grand nombre de thread. Il est alors possible de connaitre son identifiant parmi tous les threads en utilisant la fonction `get_global_id`. La variable `idx` contient ainsi un identifiant unique pour chaque thread. À partir de cette valeur, il est possible d'effectuer un calcul qui en dépend comme, par exemple, l'affectation d'une opération dans une case de tableau. Ce bout de code présente ainsi un flot de contrôle avec plusieurs chemins (instructions `if` et `else`). Mais les instructions présentes sur chacun des chemins ne diffèrent pas uniquement sur les valeurs. En effet, même s'il s'agit dans les deux cas d'une multiplication, il est nécessaire de manipuler les tableaux `A`, `B` et `D` sur le premier chemin alors qu'uniquement le tableau `C` est utile dans la branche `else`. La technique de `if-conversion` consiste alors à initialiser des prédicats en fonction d'une opération booléenne (par exemple en effectuant une comparaison). Le Listing 2.5 présente ainsi notre exemple une fois la transformation de if-conversion effectuée. Le prédicat `P1` est vrai quand le test est évalué à vrai et les instructions qui en résultent sont ainsi gardées par ce prédicat. A l'inverse, les instructions présentes dans l'autre branche sont gardées par le prédicat `P2` qui prend la valeur opposée. Cette nouvelle représentation permet ainsi d'obtenir un unique bloc de base sur lequel il est possible d'appliquer les transformations classiques pour la vectorisation. Il est juste nécessaire de faire attention aux nouvelles dépendances entre l'écriture du prédicat et ses utilisations : il s'agit de dépendances de données.

Pour étendre le domaine d'utilisation du SLP, SHIN, HALL et CHAME ont ajouté le support du flot de contrôle[70] avec les Listings de 2.7 à 2.11. Plus précisément sur le Listing 2.7 nous avons le code scalaire d'une boucle contenant un test dépendant de l'itérateur `i` et à l'intérieur du `if` nous avons deux instructions. Ensuite nous pouvons voir sur le Listing 2.8 le code après if-conversion, c'est-à-dire lorsque l'instruction `if` à été transformée en prédicat `pT`. Après cela sur le Listing 2.9 nous observons le code parallélisé après un un SLP, cette propriété est visible avec la notation suivante : nous passons de `back_blue[i] = fore_blue[i];` à `back_blue[i:i+3] = fore_blue[i:i+3];`, cela signifie que les cases de i à i+3 vont êtres traitées simultanément. De plus, à cause de la dépendance de `back_red[i]` pour `back_red[i+1]`,

nous sommes obligés de repasser du vectoriel au scalaire et donc dépaqueter le vecteur `v_pT`. L'étape suivante visible sur le Listing 2.10 consiste à remplacer le prédicat `(v_pT)` par un « select ». Enfin nous « dé-prédicatons » c'est-à-dire nous remplaçons de nouveau les prédicats par des `if` sur le Listing 2.11

Listing 2.7 – code d'origine[70]

```
1  for (i=0; i<1024; i++){
2      if(fore_blue[i] != 255){
3          back_blue[i] = fore_blue[i];
4          back_red[i+1] = back_red[i];
5      }
6  }
```

Listing 2.8 – code if converté [70]

```
1  for(i=0; i<1024; i+=4){
2          comp = fore_blue[i] != 255;
3          pT, pF = pset(comp);
4  (pT)    back_blue[i] = fore_blue[i];
5  (pT)    back_red[i+1] = back_red[i];
6  ...
7  }
```

Listing 2.9 – code parallélisé [70]

```
1  for(i=0; i<1024; i+=4){
2          v_comp =
3              fore_blue[i:i+3]
4              !=(255,255,255,255);
5          v_pT, v_pF = v_pset(v_comp);
6  (v_pT)  back_blue[i:i+3] = fore_blue[i:i+3];
7          pT1, pT2, pT3, pT4 = unpack(v_pT);
8  (pT1)   back_red[i+1] = back_red[i];
9  (pT2)   back_red[i+2] = back_red[i+1];
10 (pT3)   back_red[i+3] = back_red[i+2];
11 (pT4)   back_red[i+4] = back_red[i+3];
12 }
```

Listing 2.10 – code avec select[70]

```
1  for(i=0; i<1024; i+=4){
2          v_comp =  fore_blue[i:i+3]
3                       !=(255,255,255,255);
4          v_pT, v_pF = v_pset(v_comp);
5          back_blue[i:i+3]=
6                 select(back_blue[i:i+3],
7                 fore_blue[i:i+3],
8                 v_pT)
9           pT1, pT2, pT3, pT4 = unpack(v_pT);
10 (pT1)   back_red[i+1] = back_red[i];
11 (pT2)   back_red[i+2] = back_red[i+1];
12 (pT3)   back_red[i+3] = back_red[i+2];
13 (pT4)   back_red[i+4] = back_red[i+3];
14 }
```

Listing 2.11 – code « dé-prédicaté »[70]

```
1  for(i=0; i<1024; i+=4){
2      v_comp =
3        fore_blue[i:i+3]!=(255,255,255,255);
4      v_pT, v_pF = v_pset(v_comp);
5      back_blue[i:i+3] =
6        select(back_blue[i:i+3],
7                  fore_blue[i:i+3],
8                  v_pT)
9      pT1, pT2, pT3, pT4 = unpack(v_pT);
10     if(pT1) back_red[i+1] = back_red[i];
11     if(pT2) back_red[i+2] = back_red[i+1];
12     if(pT3) back_red[i+3] = back_red[i+2];
13     if(pT4) back_red[i+4] = back_red[i+3];
14 }
```

SHIN [67] a créé les Branches-On-Superword-Condition-Codes (BOSCC). Il s'agit d'une instruction conditionnée qui peut être basée sur le résultat de la comparaison de deux vecteurs ce qui sert à réinjecter du flot de contrôle après la if-conversion pour gérer les cas ou un masque de prédicat est nul, c'est-à-dire lorsque l'instruction à effectuée ne sera jamais exécuté. Pour visualiser cela, le code scalaire du Listing 2.12 introduit un simple `if` testant si la case courante de `fore` n'est pas 255. Dans ce cas, on écrase la valeur de la case `i` de `back` par celle de `fore`. Ensuite nous réalisons une if-conversion sur le Listing 2.13 pour des vecteurs de taille 4, c'est-à-dire (i) utilisation d'un vecteur constant rempli de 255 et (ii) création d'un vecteur pour stocker le résultat du test `if(fore[i] != 255)` (devenu `v_pT = fore[i:i+3] != v255;`). Ce résultat est utilisé avec le `select`. C'est dans la dernière étape que l'on utilise les BOSCC, les changements sont visibles

dans les deux lignes surlignées. Ainsi nous n'utiliserons le select que lorsque le vecteur `v_pT` n'est pas vide, c'est-à-dire lorsque pour une itération donnée les quatre case de `fore` ne valent pas 255.

Listing 2.12 – code séquentiel[69]

```
1 for(i=0;i<1024;i++)
2     if(fore[i] != 255)
3         back[i] = fore[i];
```

Listing 2.13 – code après if-conversion[69]

```
1 for(i=0;i<1024;i+=4){
2     v255 = {255,255,255};
3     v_pT = fore[i:i+3] != v255;
4     back[i:i+3] = select(back[i:i+3],fore[i:i+3],v_pT)
5 }
```

Listing 2.14 – code avec un BOSCC[69]

```
1 for(i=0;i<1024;i+=4){
2     v255 = {255,255,255};
3     v_pT = fore[i:i+3] != v255;
4     branch-on-none(v_pT) LI;
5     back[i:i+3] = select(back[i:i+3],fore[i:i+3],v pT)
6     LI:
7 }
```

Gestion de flot de contrôle imbriqué

Le flot de contrôle imbriqué pose de nouveaux problèmes ne pouvant être résolus avec les techniques précédentes. En effet, on observe une potentielle explosion des chemins, au pire de l'ordre de 2^n, n le nombre de tests, donc exponentiel. Mais aussi l'approche optimiste atteints des limites pour la même raison, si nous voulons générer des prédicats pour tous les branchements cela devient rapidement trop coûteux.

Pour pallier cela, il faut sélectionner les branchements à favoriser. Sur le Listing 2.15 nous pouvons voir un code scalaire contenant du flot de contrôle imbriqué, deux `if` dans un autre `if`. Lorsque nous traduisons le code scalaire en code pour le réécrire avec des BOSCC nous obtenons le Listing 2.16. Sur celui-ci nous avons la suite d'instructions avec soit la construction de vecteur (pour les initialiser soit pour leur faire stocker le résultat d'une comparaison). Le problème est que, dans ce cas, tous les tests vont être évalués, quel que soit le jeu d'entrée. C'est pour cela que SHIN[67] a ensuite voulu rendre les BOSCC plus performants en supportant leurs imbrications. En effet comme cela est visible sur le Listing 2.17 `if (B1 == 1)` (correspondant au test de la boucle for) et `if (B3 == 1)` (correspondant aux `if(xx<8)` du Listing 2.15) englobe désormais les deux derniers `if` permettant dans certains cas d'obtenir un gain de temps.

Listing 2.15 – Boucle scalaire [67]

```
for(i=0; i<SIZE; i++){
    xx = in[i];
    if(xx<8){
        if(xx==0){
            out[i] = 0;
            goto end;
        }
        if(xx<0){
            out[i] = 2;
            goto end;
        }
        out[i] = xx+xx;
    }
    end:;
}
```

Listing 2.16 – Boucle avec les BOSCC single level[67]

```
L:      ...
   V3 = vec cmplt(V1, V2);
   B1 = vec any ne(V3, vF);
   if (B1 == 1){... }...
   V4 = vec cmpeq(V1, vZero);
   V5 = select(vZero, V4, V3);
   B2 = vec any ne(V5, vF);
   if (B2 == 1){... }...
   B4 = vec any ne(V7, vF);
   if (B4 == 1){... }...
   B5 = vec any ne(V8, vF);
   if (B5 == 1){... }...
   B3 = vec any ne(V6, vF);
   if (B3 == 1){... }...
   goto L;
```

Listing 2.17 – Boucle avec les BOSCC imbriqués [67]

```
L:      ...
  B1 = vec any lt(V1, V2);
  if (B1 == 1){
     V3 = vec cmplt(V1, V2);...
     V4 = vec cmpeq(V1, vZero);
     V5 = select(vZero, V4, V3);
     B2 = vec any ne(V5, vF);
     if (B2 == 1){... }...
     B3 = vec any ne(V6, vF);
     if (B3 == 1){...
       B4 = vec any ne(V7, vF);
       if (B4 == 1){... }...
       B5 = vec any ne(V8, vF);
       if (B5 == 1){... }}}...
  goto L;
```

2.2.2 Parallélisme vectoriel d'un nid de boucle

Vectorisation dans un nid de boucle parfait uniquement

Après avoir discuté sur les manières de faire apparaitre la vectorisation dans des boucles non imbriquées (de profondeur un) attardons-nous sur les techniques permettant d'apporter la vectorisation dans des nids de boucles. Le FORTRAN 8x a introduit par rapport à la norme précédente la notion « d'array notation ». Celle-ci a été mise en place pour les supercalculateurs vectoriels (typiquement les machines Cray) par ALLEN et KENNEDY [1, 2]. En effet comme nous pouvons le voir le code du Listing 2.18 contenant deux boucles imbriquées est traduit sur le Listing 2.19 en une simple ligne utilisant le caractère « : », celui-ci permettant d'opérer l'instruction sur l'ensemble des cases entre les deux bornes.

Listing 2.18 – Nid de boucle classique en FORTRAN [2]

```
   DO 10, I = 1,100
      DO 20, J=1,100
         X(I,J) = X(I,J) + Y(I,J)
   20 CONTINUE
10 CONTINUE
```

Listing 2.19 – Boucle vectorisé avec des « array notations » en FORTRAN 8x[2]

```
X(1:100,1:100) = X(1:100,1:100) + Y(1:100,1:100)
```

Cette « array-notation » permet d'exprimer simplement des instructions difficilement exprimables avec des boucles. Nous avons sur le Listing 2.20 un cas complexe que nous cherchons à vectoriser en FORTRAN et nous pouvons voir sur le Listing 2.21 que nous avons pu le faire en une seule ligne. Un autre exemple complexe est illustré sur le Listing 2.22 où nous avons l'instruction `WHERE` qui permet de faire `A = A + B` quand `A` est plus grand que `0`, si nous avions voulu l'écrire autrement nous aurions du utilisé du flot de contrôle.

Listing 2.20 – Exemple plus complexe en FORTRAN, ce que l'on veut faire [2]

```
1 A(1, 1) = B(1, J) + C(1, 3)
2 A(1, 2) = B(2, J) + C(1, 4)
3           ...
4 A(1, M) = B(M, J) + C(1, M + 2)
```

Listing 2.21 – Exemple plus complexe en FORTRAN 8x, avec des « array notations »[2]

```
1 A(1, 1:M) = B(1:M, J) + C(1, 3:M + 2)
```

Listing 2.22 – Exemple d'approche de la notion de masque avec des « array notations » en FORTRAN 8x[2]

```
1 WHERE(A .GT. 0.0) A = A + B
```

Certes, les « array notations » permettent d'avoir un code simplifié et plus performant grâce à la vectorisation introduite. Mais pour les exploiter il faut modifier à la main les anciens codes, c'est pour cette raison que ALLEN et KENNEDY [1] ont développé un outil source à source prenant en entrée un ancien code FORTRAN et sortant un code en FORTRAN 8x. Nous pouvons voir le fonctionnement sur le Figure 2.3 : sur la gauche, nous avons l'analyseur qui, par exemple, prendrait le code du Listing 2.18 ou Listing 2.20. Après il est traduit pour aboutir sur les codes du type Listing 2.19, Listing 2.21 ou encore Listing 2.22.

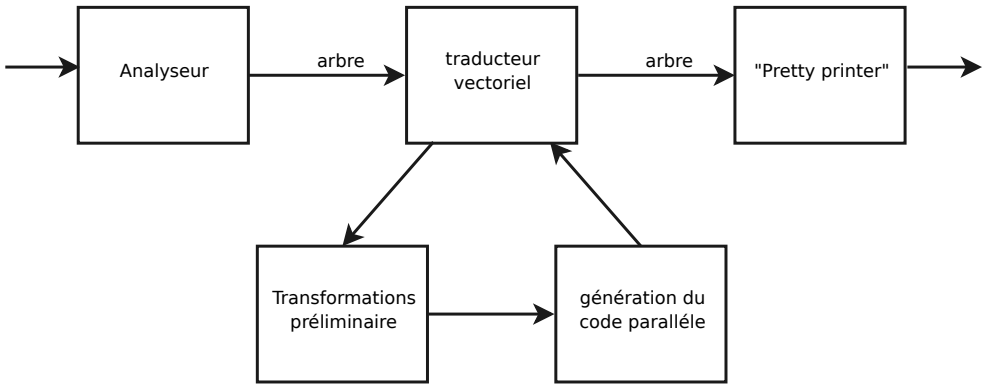


FIGURE 2.3 – Fonctionnement du traducteur Fortran vers Fortran 8x [2].

Prise en charge des nids de boucles non parfaits

Les nids de boucle des codes de production sont généralement plus complexes : ils peuvent contenir des instructions ailleurs que dans la boucle la plus profonde et aussi du flot de contrôle à l'intérieur ce qui n'est pas géré par les techniques précédentes.

Pour abstraire la vectorisation de boucle irrégulière WU et al. [75] ont proposé la notion de « vecteur virtuel ». Celle-ci est définie comme suit : c'est une abstraction des vecteurs fournis par la machine au travers des registres SIMD. Nous avons la Figure 2.4 qui indique les trois étapes du fonctionnement :

— l'agrégation pour extraire du parallélisme SIMD à différend niveau dans un programme qui se divise en 3 parties (i) l'agrégation au niveau des blocs de base pour extraire du parallélisme SIMD au sein

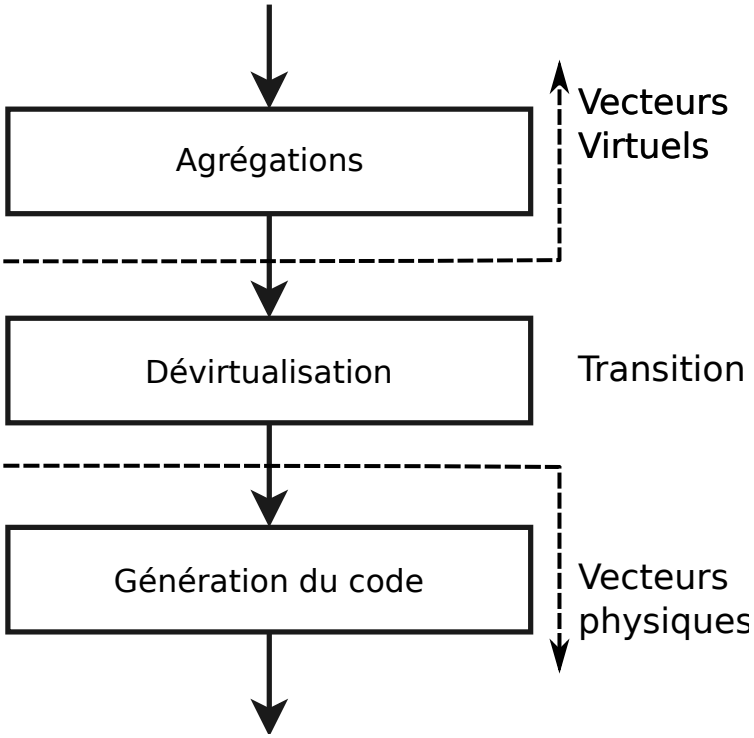


FIGURE 2.4 – Fonctionnement simplifié du framework de SIMDisation utilisant les vecteurs virtuels[75]

des blocs, (ii) l'agrégation des petites boucles pour éliminer les boucles trop simples (un petit et connu à la compilation nombre d'itérations) et (iii) l'agrégation au niveau des boucles pour extraire du parallélisme SIMD parmi les itérations, tout ceci permet de générer des vecteurs virtuels.

— dévirtualisation qui se déroule en deux phases : (i) la dévirtualisation de l'alignement qui réaligne les données dans les registres et (ii) la dévirtualisation qui permet d'obtenir les vecteurs physiques

— génération du code qui va mapper les opérations en instructions dépendantes de la cible (intrinsics par exemple)

Un exemple de nid de boucle irrégulier exploitable est visible sur la Listing 2.23. Ici, nous avons deux boucles imbriquées avec une réduction, donc elle est non-parfaite.

Listing 2.23 – Exemple de code pour les vecteurs virtuels

```
1 short input[], coef[];
2 for (i=0 ; i<NInput ; i++){
3         int sum=0;
4         for (int j=0 ; j<16 ; j++)
5                 sum+=input[k+j] * coef[j];
6         output[i]=sum;
7 }
```

La récursivité (on parle aussi du caractère « Deep ») des techniques développées dans cette section pour la vectorisation de nid de boucle irrégulier est un critère très important pour nous. C'est avec cela comme principe que XU, SUN et ZHAO ont crée Codegen puis plus tard SimdCodegen[76]. Leurs points communs sont le fait d'appliquer récursivement du Strip Mining dans la limite du possible avec comme objectif le cassage des dépendances cyclique qui entravent la vectorisation. Leurs différences sont que Codegen a été conçu pour les machines vectorielles et que simdcodegen est son extension pour les coprocesseurs SIMD tels

que le MIC. Sur la Figure 2.5 on peut voir sur la gauche une boucle qui sert d'exemple scalaire en FORTRAN et nous pouvons voir que pour l'instruction S2 il y a une dépendance envers S1. En effet si nous prenons un `i` valant 1 il y a une dépendance entre `B[i+4]` et `B[i]`. Au milieu, nous avons deux boucles imbriquées résultant du Strip Mining avec un stride de 4. En effet, la distance entre les cases liées par les instructions est de 4. Tout à droite nous avons le nid de boucle parallélisé avec les « array notation » et cette fois-ci sans dépendance.

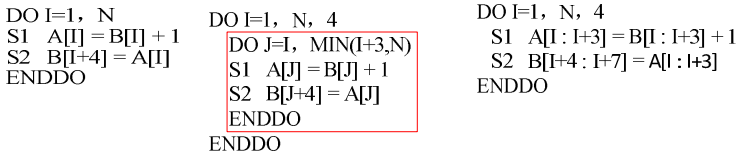


FIGURE 2.5 – Example de rupture de cycle[76]

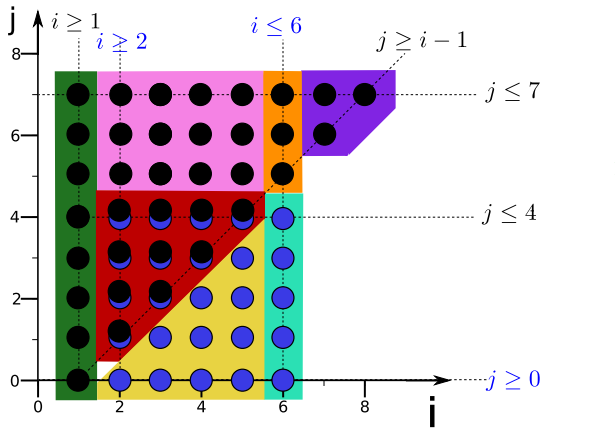
Pour paralléliser un nid de boucles irrégulier, nous pouvons aussi nous appuyer sur une autre technique existante, le modèle polyédrique utilisant l'algèbre linéaire, la géométrie ou encore du calcul matriciel pour réordonnancer le corps d'un nid de boucle[5][8]. Pour visualiser son fonctionnement, nous allons détailler un exemple : pour le code scalaire visible sur le Listing 2.24 qui contient deux instructions `S1` et `S2`, nous pouvons représenter le nid de boucle sur le Tableau 2.1a. Sur cette figure, les points noirs correspondent aux valeurs prisent par `S1` et les points bleus ceux pris par `S2`, nous avons donc un plan, car les instructions dépendent de deux index (i et j sur le Listing 2.24). Une fois redécoupé, ce domaine est divisé en 7 sous domaines qui seront redécoupés et retraduits en code sur le Listing 2.25.

Nous pouvons noter que dans cette section nous n'avons pu trouver de technique qui gérerait à la fois des nids de boucles non parfaits et du flot de contrôle imbriqué dans ce nid.

2.2.3 Vectorisation d'une fonction, SPMD

Nous avons une autre approche de la vectorisation se basant sur OpenCL Listing 2.27. Nous sommes face à du SPMD (Single Progam Multiple Data) et non du SIMD, c'est-à-dire que la parallélisation ne s'exprime plus au travers d'une boucle, mais plutôt au travers d'une fonction. En fait, comme nous l'avons expliqué précédemment, cette fonction (« kernel ») sera exécutée par un grand nombre de threads simultanément. Nous pouvons « éclater » un nid de boucle pour mettre son contenu dans une fonction en pouvant exprimer l'itérateur grâce aux identifiants des threads. Sur le Listing 2.26 [2] nous avons un exemple d'une fonction séquentielle qui réalise une multiplication de deux vecteurs dans un troisième. Une fois réécrite en OpenCL, nous avons le kernel du Listing 2.27 sur lequel on notera la suppression de la boucle et l'utilisation à la place du `get_global_id(0)` qui récupère l'identifiant du thread courant.

2. `https://en.wikipedia.org/wiki/OpenCL`



(a) Représentation géométrique

Listing 2.24 – code d'origine

```
1  for (i=1;i≤8;i++) {
2      for (j=i-1;j≤7;j++) {
3          S1(i,j);
4      }
5      if ((i≥2)&&(i≤6)) {
6          for (j=0;j≤4;j++) {
7              S2(i,j);
8          }
9      }
10 }
```

Listing 2.25 – code transformé

```
1  for (j=0;j≤7;j++) {
2   S1(1,j);
3  }
4  for (i=2;i≤5;i++) {
5   for (j=0;j≤i-2;j++)
6       S2(i,j);
7   for (j=i-1;j≤4;j++) {
8       S1(i,j);
9       S2(i,j);
10  }
11  for (j=5;j≤7;j++)
12      S1(i,j);
13 }
14 for (j=0;j≤4;j++)
15     S2(6,j);
16 for (j=5;j≤7;j++)
17     S1(6,j);
18 for (i=7;i≤8;i++) {
19  for (j=i-1;j≤7;j++) {
20      S1(i,j);
21  }
22 }
```

TABLE 2.1 – Exemple d'application du modèle polyédrique

Listing 2.26 – kernel séquentiel de multiplication de deux vecteurs

```
1  void array_mul(int n,
2                 const float *a,
3                 const float *b,
4                 float       *c)
5  {
6    int i;
7    for (i = 0; i < n; i++)
8      c[i] = a[i] * b[i];
9  }
```

Listing 2.27 – kernel en OpenCL de de multiplication de deux vecteurs

```
1  __kernel
2  void scalar_mul(
3    __global const float *a,
4    __global const float *b,
5    __global float *c)
6  {
7          int id = get_global_id(0);
8          c[id] = a[id] * b[id];
9  }
```

Grâce au langage OpenCL qui permet une expression vectorielle explicite, KARRENBERG et HACK ont implémenté une transformation qui est à la fois plateforme et langage indépendant pour faire de la vectorisation. Cela à partir de la représentation intermédiaire du type graphe de flot de contrôle SSA et en tant qu'une passe dans un driver OpenCL et aussi dans un code de lancer de rayon. Le fonctionnement consiste à l'utilisation des masques pour abstraire le flot de contrôle (if conversion) en ciblant les architectures n'ayant pas de sup-

port matériel d'instructions prédicatées[34]. D'un autre point de vue nous pouvons analyser les techniques d'optimisation qui sont là pour faciliter la vectorisation par le compilateur.

2.3 Transformation pour faciliter la vectorisation

Les algorithmes de vectorisation peuvent avoir besoin de certaines transformations pour permettre la génération d'un meilleur code. Par exemple, une transformation manuelle extrême peut aider le compilateur. Celle-ci se nommant Nija Gap qui est, selon KIM et al., l'écart entre un code "inconscient du parallélisme" (version séquentielle généralement) et de sa version la mieux optimisée [36].

Selon une autre approche, pour minimiser la synchronisation entre les boucles à fusionner tout en maximisant le parallélisme KENNEDY et MCKINLEY ont étudié la fusion de boucle pour les codes en FORTRAN, celle-ci pouvant favoriser la localité [35]. En effet nous pouvons voir dans le Listing 2.28 deux boucles avec le même domaine d'itération. La première contenant une instruction écrivant dans `A(I)` et la seconde contenant une instruction lisant `A(I)`. Donc, si nous fusionnons les deux boucles nous obtenons la boucle visible sur le Listing 2.29 dans laquelle le compilateur va pouvoir mieux optimiser l'utilisation du cache. Pour les boucles de calcul conséquentes une meilleure localité favorise le SLP et l'utilisation des registres.

Listing 2.28 – Boucles classique en FORTRAN [35]

```
1 PARALLEL DO I = 1, N
2     A(I)  = 0.0
3 END PARALLEL
4 PARALLEL DO I  = 1, N
5     B(I) = A(I)
6 END PARALLEL
```

Listing 2.29 – Boucle après fusion [35]

```
1 PARALLEL DO I  = i, N
2     A(I) = 0.0
3     B(I) = A(I)
4 END PARALLEL
```

Ainsi, vis-à-vis du flot de données, KIM et HAN ont cherché à régulariser des accès à des tableaux grâce à la réorganisation du corps des boucles irrégulières. En effet l'indirection amène divers problèmes lors de la vectorisation, tels que des alignements inconnus ou encore des dépendances cycliques[37]. Les auteurs ont proposé une méthode de compilation pour vectoriser des boucles contenant des tableaux avec accès indirect. En effet nous pouvons voir sur la Figure 2.6 les étapes pour réaliser leur méthode de vectoriser : (i)une phase de preprocessing où des if-conversion vont être réalisés pour transformer le flot de contrôle en flot de données, (ii) construction du graphe de flot de de données (DFG en anglais) où chaque arc exprime la dépendance entre deux opérations, (iii) différentiation pour chaque opération si elle est vectorisable ou non, (iv) ajout de code pour faire du « unpacking » ou du « packing » dans des registres vectoriels pour distribuer ou faire l'inverse des données non contiguës en mémoire, et cela, à l'aide d'heuristiques, (v)extraction du parallélisme intra opération (SLP voir la section 2.2.1) des opérations scalaires restantes et (vi) génération du code vectorisé à partir du DFG modifié.

Une technique pour minimiser automatiquement le nombre d'opérations de récupération de données dans le but de satisfaire les alignements a été étudiée par EICHENBERGER, WU et O'BRIEN [23]. En premier voyons un code avec des références mal-alignées, pour cela nous avons le Listing 2.30 représentant l'addition de deux vecteurs, mais au lieu d'avoir simplement `a[i] = b[i] + c[i];` nous avons des accès à des indices qui nous permet de mettre en lumière des dépence inter-itérations telles qu'entre `a[i]` et `a[i+3]`. Sur la Figure 2.7 nous avons la manière dont a été modifié le listing précédent pour le vectoriser. Plus précisément sur la Figure 2.7b nous avons le flux mémoire de l'instruction `b[i+1]` de 0 à 99, c'est à dire de `b[1]` à `b[100]`. De la même manière, nous avons la Figure 2.7c pour l'instruction `c[i+2]` variant de `c[2]` à `c[101]` et la Figure 2.7d pour `a[i+3]`. Sur la Figure 2.7a nous pouvons constater l'ajout d'un nouvel opérateur

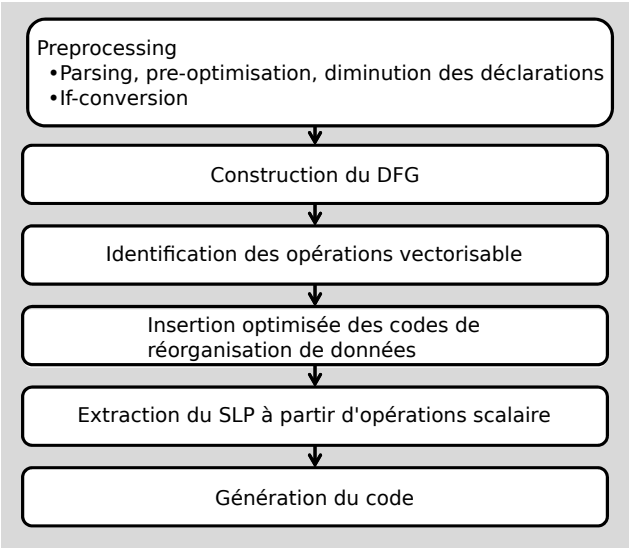


FIGURE 2.6 – Fonctionnement de l'algorithme [37]

pour la réorganisation de données, `vshiftstream(c1, c2)`, qui prend un flux de registre à l'offset c_1 pour générer un flux de registre avec les mêmes valeurs, mais à l'offset c_2. Ainsi la « simdisation » devient valide.

Listing 2.30 – Exemple de boucle [23]

```
1 for (i = 0; i < 100; i++) {
2     a[i+3] = b[i+1] + c[i+2];
3 }
```

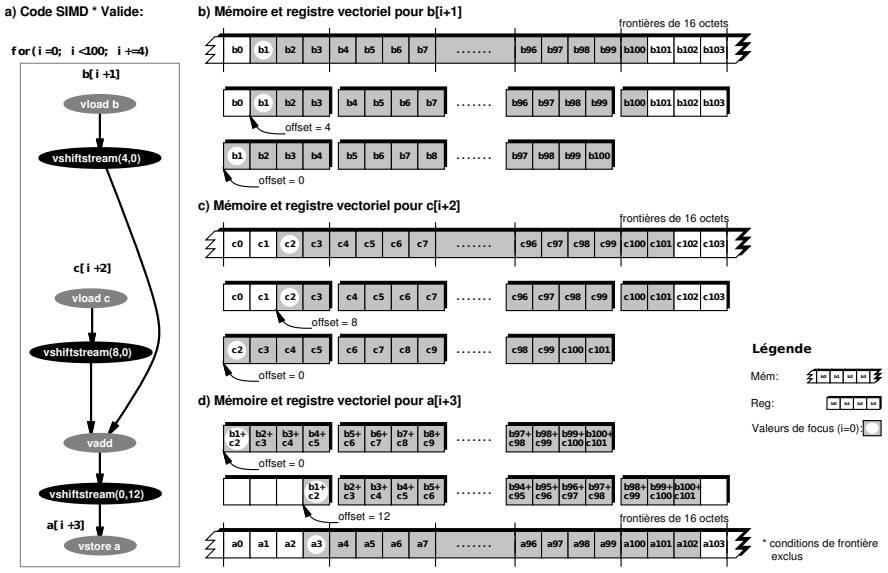


FIGURE 2.7 – Simdisation valide pour les architectures supportant des contraintes d'alignement [23]

Si nous revenons sur la puce décrite dans le chapitre précédent, le Xeon Phi, il y a de nouvelles méthodes ont été conçues pour optimiser l'utilisation des registres SIMD par TIAN et al.[72], dont celle qui nous intéresse

l'optimisation « less-than-full-vector loop vectorization » qui permet de transformer le code du Listing 2.31 en celui du Listing 2.32. Plus précisément pour le code séquentiel nous voyons que le nombre d'itérations n'est pas connu à la compilation, mais aussi l'alignement du pointeur `y` ne l'est pas non plus. Du côté du code optimisé, nous commençons par assurer l'alignement de y à 64 octets (car les registres sont de 512 bits) puis nous exécutons les itérations non alignées (« peeling loop ») donc non vectorisables. Ensuite le reste du code est vectorisé grâce aux fonctions intrinsèques.

Listing 2.31 – code séquentiel[72]

```
float foo(float *y, int n){
     int k;
     float x = 10.0f;
     for (k = 0; k < n; k++) {
         x = x + fsqrt(y[k])
    }
    return x;
}
```

Listing 2.32 – pseudo code avec la technique de vectorisation « less-than-full-vector loop vectorization »[72]

```
misalign = &y[0] & 63
peeledTripCount = (63 - misalign)/sizeof(float) x = 10.0f;
do k0=0, peeledTripCount-1  // peeling loop
    x=x + fsqrt(y[k0])
enddo
x1_v512 = (m512)0
x2_v512 = (m512)0
mainTripCount = n - ((n - peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps(
                    _mm512_fsqrt(y[k1:16]),x1_v512)
  x2_v512 = _mm512_add_ps(
                    _mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// perform vector add on two vector x1_v512 and x2_v512
x1_v512 = _mm512_add_ps(x1_v512, x2_512);
// perform horizontal add on all elements of x1_v512, and
// the add x for using its value in the remainder loop
x = x + _mm512_hadd_ps(x1_512)
do k2 = mainTripCount, n  // Remainder loop
   x = x + fsqrt(y[k2])
enddo
```

Comme nous l'avons vu au chapitre précédent les processeurs x86_64 ne sont pas les seuls à avoir des fonctions intrinsics. En effet, sur l'architecture ARM nous avons à notre disposition l'ABI NEON [3] . Pour ne pas a avoir à exprimer directement la vectorisation, une possibilité est de passer par un langage nativement vectoriel, par exemple l'OpenCL. Jo et al. ont conçu un driver OpenCL pour faire cela[31]. Sur le Listing 2.33 nous avons un petit kernel écrit en OpenCL, il réalise la multiplication de deux vecteurs. Après nous avons une première manière de traduire ce code en réinsérant des boucles, mais en gardant les types OpenCL sur le Listing 2.34. À partir de là, une autre représentation est détaillée sur le Listing 2.35 cette fois-ci en utilisant les intrinsics NEON.

Listing 2.33 – kernel OpenCL[31]

```
__kernel void foo(__global int4* dst, __global int4* src) {
    int id = get_global_id(0);
    dst[id] = src[id] * src[id];
}
```

3. http://www.arm.com/products/processors/technologies/neon.php

Listing 2.34 – Au niveau d'un « work group » avec des opérations vectorielles Jo :2014 :OFA :2568058.2568064

```
void foo(int4* dst, int4* src) {
    int id;
    for (__k = 0; __k < __local_size[2]; __k++) {
        for (__j = 0; __j < __local_size[1]; __j++) {
            for (__i = 0; __i < __local_size[0]; __i++) {
                dst[id] = (int4){src.v[0]*src.v[0], src.v[1]*src.v[1],
                                 src.v[2]*src.v[2], src.v[3]*src.v[3]}
            }
        }
    }
}
```

Listing 2.35 – Au niveau d'un « work group » avec intrinsic[31]

```
void foo(int4* dst, int4* src) {
    int id;
    for (__k = 0; __k < __local_size[2]; __k++) {
        for (__j = 0; __j < __local_size[1]; __j++) {
            for (__i = 0; __i < __local_size[0]; __i++) {
                dst[id].neon = vmulq_s32(src[id].neon, src[id].neon);
            }
        }
    }
}
```

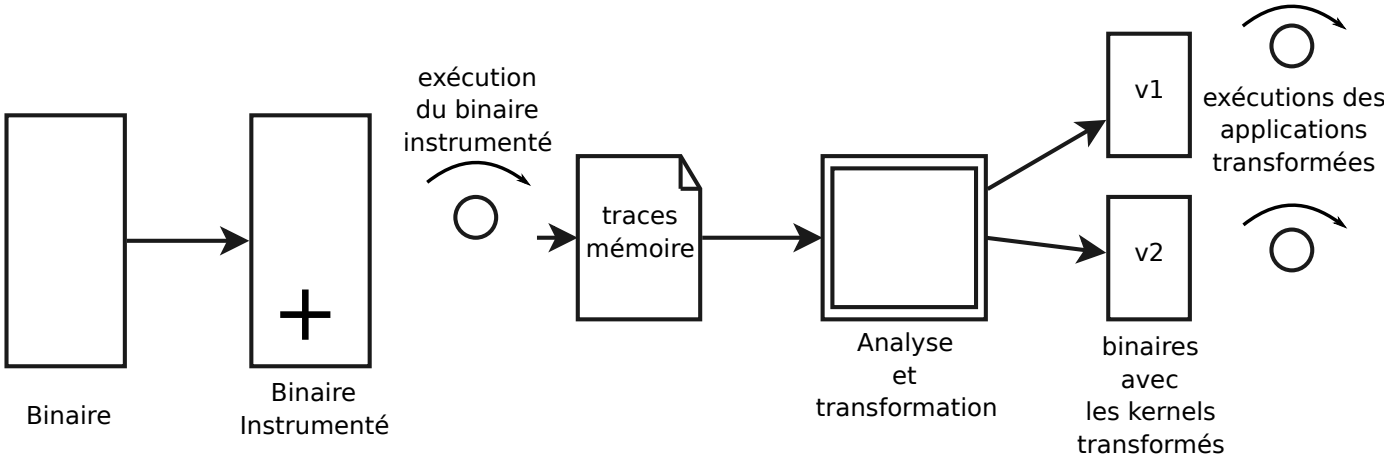


FIGURE 2.8 – Fonctionnement de l'algorithme [27].

Étudions une autre manière de travailler, cette fois nous sommes au niveau du binaire et non de la compilation. Plus précisément HAINE et al. se sont servis d'un outil d'analyse de binaire MAQAO [4] qui leur permet d'avoir une bonne estimation des transformations à effectuer pour améliorer les performances[27]. En effet, il peut donner au programmeur un bon aperçu des transformations faisable pour améliorer la vectorisation et cela grâce à un algorithme visible sur la Figure 2.8. Celui-ci commence par instrumenter le binaire pour générer des traces en le faisant tourner, ensuite celles-ci sont réinjectées pour servir d'input à l'outil d'analyse pour appliquer les modifications et enfin faire exécuter le nouveau binaire pour s'assurer de la pertinence des optimisations effectuées.

Enfin, le Deep Jam[15] est une technique conçue à l'origine pour convertir du parallélisme à gros grains vers du parallélisme à grain fin (au niveau des instructions). Son fonctionnement est détaillé sur la Figure 2.10 :

4. http://maqao.org

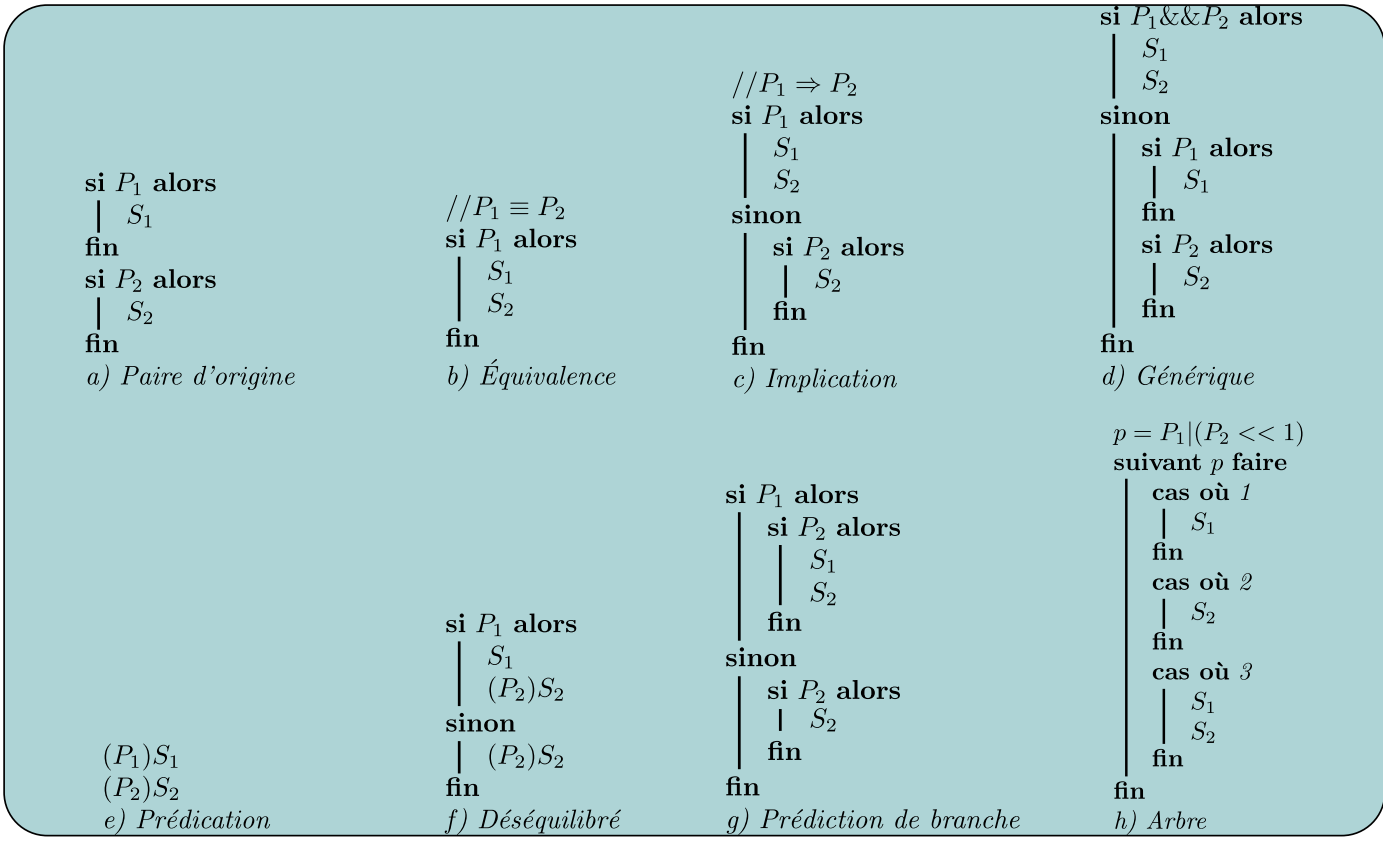


FIGURE 2.9 – Exemple d'heuristiques pour une paire if-if [15].

(i) Strip Mining (voir la section 3.1.3) selon un facteur nommé « facteur du Deep Jam » (ii) création de l'arbre de dépendance pour associer les instructions opérant sur les mêmes variables et ainsi supprimer les dépendances redondantes, (iii) découpage en threadlets (petit morceau de code), (iv) fusion (jamming) pour toutes les paires de threadlets à l'aide d'heuristiques (des exemples sont visibles sur la Figure 2.9 pour une paire « if-if ») pour réécrire l'arbre des dépendances et (v)génération du code optimisé. Même si le Deep Jam est une technique pas directement conçue pour le parallélisme SIMD, elle permet tout de même de rapprocher des instructions entre elles.

2.4 Conclusion

Dans ce chapitre, nous avons décrit les travaux existants autour de la vectorisation et de l'exploitation des unités de calcul vectoriel à partir d'un code source. Dans le cas d'un flot de contrôle irrégulier et d'un accès aux données complexe, il existe un manque dans la littérature pour transformer ce genre de nid de boucles. Dans cette thèse, nous formulons la proposition de mettre au point une telle transformation, en partant de la méthode la plus flexible pour traiter les codes complexes : le Deep Jam.

Maintenant, regardons comment nous avons étendu le Deep Jam pour faire apparaitre le parallélisme vectoriel au lieu du parallélisme d'instructions.

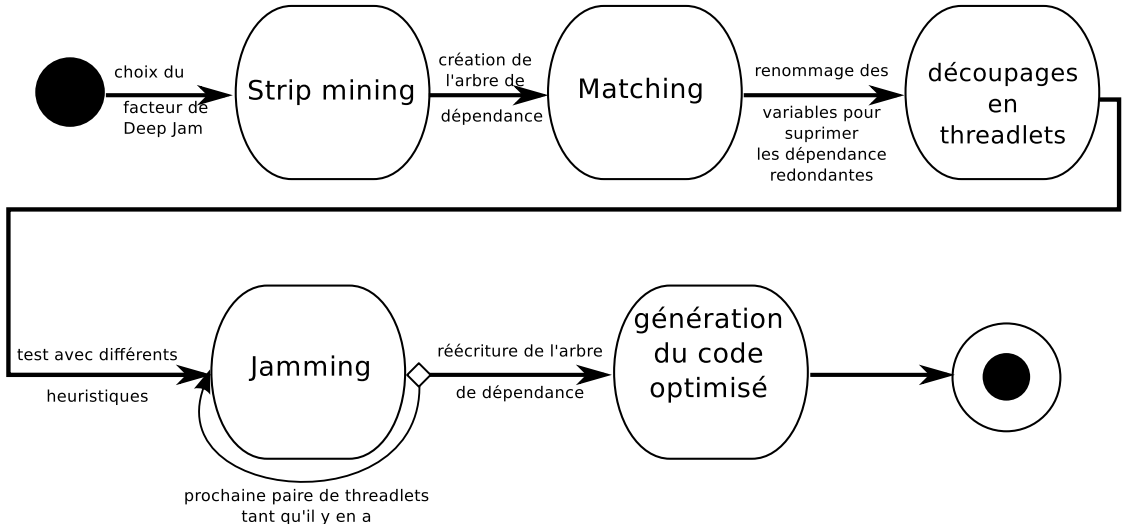


FIGURE 2.10 – Algorithme du Deep Jam

Chapitre 3

Extension de la transformation Deep Jam pour la vectorisation

Comme dit précédemment, la transformation Deep Jam permet d'extraire du parallélisme à grain fin, mais qui reste à adapter à notre besoin principal : faire apparaitre la vectorisation dans un ou plusieurs nids de boucle indépendante et contenant au pire du flot de contrôle et de données irrégulier et imbriqué. Ceci est d'autant plus intéressant qu'il y a un manque dans la littérature : faire un algorithme « deep » sur n'importe quel niveau dans un nid de boucles tout en supportant le flot de contrôle irrégulier. C'est donc avec ce double objectif que nous avons axé notre réflexion. Pour l'adaptation désirée, nous avons besoin d'un algorithme pour déterminer quand et comment appliquer le Deep Jam. Celui-ci sera explicité à l'aide d'un graphe orienté. Nous commencerons à détailler les transformations qui seront les briques de base pour définir l'algorithme. Celui-ci aura des boites noires que nous expliciterons à l'aide de microbenchmarks.

3.1 Catalogue de transformations

Pour parler du fonctionnement du Deep Jam étendu, il est nécessaire de détailler différentes transformations de code que nous utiliserons plus tard dans l'algorithme de la Figure 3.2 [3]. En effet, notre algorithme est constitué d'une suite de transformations conditionnées.

3.1.1 Hoisting

Le hoisting est l'extraction d'invariants de boucle, c'est à dire des instructions ne dépendant pas de l'itérateur ni de variables qui en dépendent dans la boucle dans laquelle elles se situent comme nous pouvons le voir avec la condition « *C* » indépendante de l'itérateur « i » sur le Listing 3.2. Elle permet, ainsi, d'avoir une boucle où le flot de contrôle disparaît, facilitant la vectorisation de la boucle :

Listing 3.1 – code initial

```
1 pour i:0→N
2     si C faire
3         A_i
4     fin si
5 fin pour
```

Listing 3.2 – code optimisé après hoisting

```
1 si C faire
2     pour i:0→N
3         A_i
4     fin pour
5 fin si
```

3.1.2 Spécialisation

Cette technique consiste en la création d'invariants de boucle par le précalcule d'une condition ce qui permet d'isoler le cas où la condition est validée pour toutes les itérations testées, cf. l'exemple du Listing 3.3 transformé en le Listing 3.4, pour, ainsi, supprimer du flot de contrôle dans ce cas là :

Listing 3.3 – code initial

```
1 pour i:0→N
2     si C_i faire
3         A_i
4     fin si
5 fin pour
```

Listing 3.4 – code optimisé après spécialisation

```
1  pour i:0→N i+=k
2      bool toutVrai ← vrai
3      pour ii : i→ i+k
4          toutVrai ← toutVrai ET C_ii
5      fin pour
6      si toutVrai faire
7          A_i
8      sinon
9          si C_i faire
10             A_i
11         fin si
12     fin si
13 fin pour
```

3.1.3 Strip Mining

Le Strip Mining est l'insertion d'une boucle interne en divisant le domaine d'itération de la boucle transformée, commoe nous pouvons voir le Listing 3.6, utile pour faire émerger une boucle avec un nombre d'itérations choisi tout en ayant la possibilité de garder l'éventuel parallélisme sur la boucle de base (et cela laisse aussi de la marge pour ajouter un nouveau niveau de parallélisme sur la boucle interne) :

Listing 3.5 – code initial

```
1 pour i:0→N
2     si C_i faire
3         A_i
4     fin si
5 fin pour
```

Listing 3.6 – code optimisé après Strip Mining

```
1 pour i:0→N i+=k
2     pour ii:i→i+k
3         si C_ii faire
4             A_ii
5         fin si
6     fin pour
7 fin pour
```

3.1.4 Tiling

Ici nous avons la généralisation multidimensionnelle du Strip Mining comme nous pouvons le voir sur le Listing 3.7 qui se transforme en le Listing 3.8. Cette transformation aide à garder des données utilisées en cache jusqu'à leurs réutilisations :

Listing 3.7 – code initial

```
1 pour i : 0→ N
2     pour j = 0→N
3         A_{i,j}
4     fin pour
5 fin pour
```

Listing 3.8 – code optimisé après tiling

```
1 pour i : 0→ N i+=2
2     pour j : 0→N j+=2
3         pour x : i→ min(i + 2, N)
4             pour y : j→ min(j + 2, N)
5                 A_{x,y}
6             fin pour
7         fin pour
8     fin pour
9 fin pour
```

3.1.5 Fusion

Cette technique est le regroupement de boucles ayant le même domaine d'itération. Ce qui peut favoriser la réutilisation des données mises en cache, dans le sens où si dans l'exemple avec les Figures 3.9 et 3.10 l'instruction B dépend de A nous aurions une meilleure utilisation du cache après fusion :

Listing 3.9 – code initial

```
1 pour i:0→N
2         A_i
3 fin pour
4 pour i:0→N
5         B_i
6 fin pour
```

Listing 3.10 – code optimisé après fusion

```
1 pour i:0→N
2         A_i
3         B_i
4 fin pour
```

3.1.6 Fission

La fission est la séparation d'une boucle en plusieurs petites, ce qui permet d'isoler des instructions, utile pour la vectorisation du fait que plus la boucle est simple plus elle a du potentiel pour les optimisations. Nous pouvons voir sur les Listings 3.11 et 3.12 un exemple :

Listing 3.11 – code initial

```
1 pour i:0→N
2         A_i
3         B_i
4 fin pour
```

Listing 3.12 – code optimisé après fission

```
1 pour i:0→N
2         A_i
3 fin pour
4 pour i:0→N
5         B_i
6 fin pour
```

3.1.7 Spliting

Cela consiste en le découpage d'une boucle en une de taille finie et l'ajout d'un « tail code » (code résiduel contenant toutes les itérations n'ayant pas été effectuées), comme cela est observable sur la transformation du Listing 3.13 pour aboutir au Listing 3.14 où une boucle de 10 itérations a été extraite :

Listing 3.13 – code initial

```
1 pour i:0→N
2         A_i
3 fin pour
```

Listing 3.14 – code optimisé

```
1 pour i:0→10
2         A_i
3 fin pour
4 pour i:10→N
5         A_i
6 fin pour
```

3.1.8 Interchange

Pour cette optimisation, nous faisons l'échange de niveaux entre deux boucles imbriquées, sur le code du Listing 3.15 la boucle itérant sur `i` et `j` vont être permutée cf. le Listing 3.16. Ceci peut être utile pour l'accès à des tableaux à plusieurs dimensions :

Listing 3.15 – code initial

```
1 pour i:0→N
2     pour j:0→M
3         A_{i,j}
4     fin pour
5 fin pour
```

Listing 3.16 – code optimisé

```
1 pour j:0→M
2     pour i:0→N
3         A_{i,j}
4     fin pour
5 fin pour
```

3.1.9 Peeling

Enfin, cette dernière technique est l'extraction d'itérations d'une boucle formant au moins une autre avec un domaine d'itération connu et fini, cf. le Listing 3.17 où le domaine d'itérations de `i` a été éclaté en 3 nous pouvons observer le résultat de cette transformation sur la Listing 3.18 :

Listing 3.17 – code initial

```
1 pour i:0→N
2         A_i
3 fin pour
```

Listing 3.18 – code optimisé

```
1 pour i:0→8
2         A_i
3 fin pour
4 pour i:8→N−30
5         A_i
6 fin pour
7 pour i:N−30→N
8         A_i
9 fin pour
```

Voyons maintenant comment ces transformations ont été exploitées pour répondre à la problématique qu'est le fait de faire émerger du parallélisme vectoriel dans du code irrégulier.

3.2 Algorithme du Deep Jam étendu

3.2.1 Exemple préliminaire d'application

Pour mieux comprendre cet algorithme, que nous allons décrire juste après, étudions un exemple : dans la Figure 3.1, nous avons un cas d'une boucle avec un nombre d'itérations fini et aucune dépendance inter

itération, ce qui nous permet d'appliquer directement le Deep Jam. Dans cet exemple nous voulons appliquer le Deep Jam sur la boucle qui itère sur `i`. Nous commençons par isoler la boucle puis réaliser un Strip Mining pour avoir une boucle interne avec un domaine d'itérations restreint de taille `k` (pour la vectorisation). Ensuite nous précalculons la condition pour extraire le cas où la condition est vraie sur tout un domaine d'itération de la boucle interne, c'est une phase de spécialisation. Viens après le hoisting où la condition C est sortie de la boucle `j` car le test ne dépend pas de l'itérateur. Enfin, nous avons la boucle interne qui contient du code, mais sans condition qui s'avère être « vectorization-friendly ».

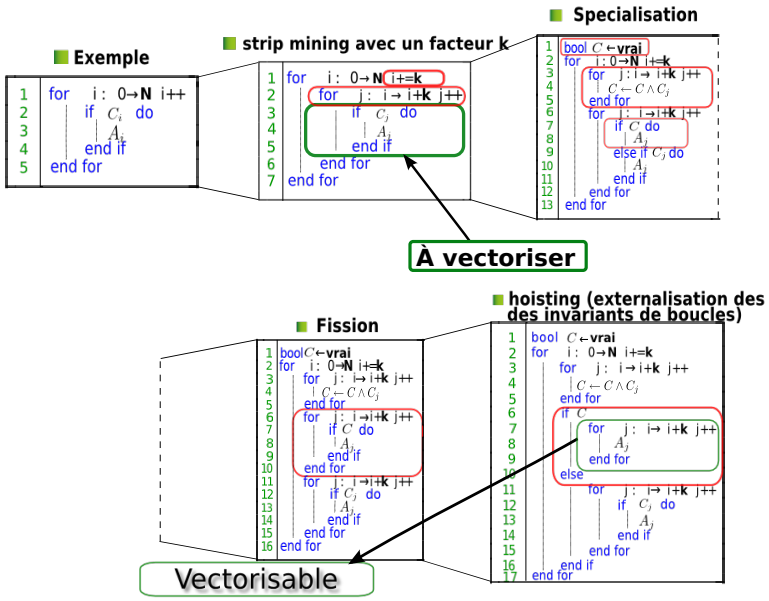


FIGURE 3.1 – Exemple d'application d'une variante du Deep Jam

3.2.2 Algorithme du Deep Jam étendu à la vectorisation de code irrégulier

Cet algorithme du Deep Jam étendu à la vectorisation débute par une phase de sélection. En effet, face à un code nous devons commencer par choisir une boucle potentiellement vectorisable, nous entendons par là les boucles de calcul qui pourrait -moyennant des transformations- s'avérer favorable à la vectorisation, par exemple une boucle avec une borne supérieure non fixe ne correspondrait pas aux critères et, si possible, avec du flot de contrôle. En effet, l'intérêt de notre variante du Deep Jam est de pouvoir l'appliquer sur des codes contenant du flot de contrôle irrégulier.

Pour un threadlet appelé sur le graphe une région Single Entry Single Exit (SESE) nous pouvons avoir affaire à la présence d'imbrications de flot de contrôle. Dans cette situation, il faudrait démultiplier les tests pour permettre de favoriser le cas où toutes les conditions imbriquées sont réunies. Mais reste à savoir jusqu'à quelle profondeur il est possible de plonger et comment choisir le nombre d'itérations à fusionner (il faudrait connaitre la taille d'un vecteur ou alors la taille d'une ligne de cache, si tant est que la solution la plus optimale soit une de ces deux possibilités)

Il faut, après, dérouler la boucle selon ce facteur à déterminer en traitant les paquets de `if` indépendants de manière séparée (s'il y en a évidemment). Face à de la récursivité de flot de contrôle, il faut choisir quels sont les chemins à privilégier.

Nous pouvons visualiser le choix des transformations utilisées à l'aide d'un digraph avec les oracles cf. la Figure 3.2, celui-ci rend visible l'aspect « deep » par sa récursivité (cycle sur le graphe) et avec en plus la gestion de flot de contrôle imbriqué.

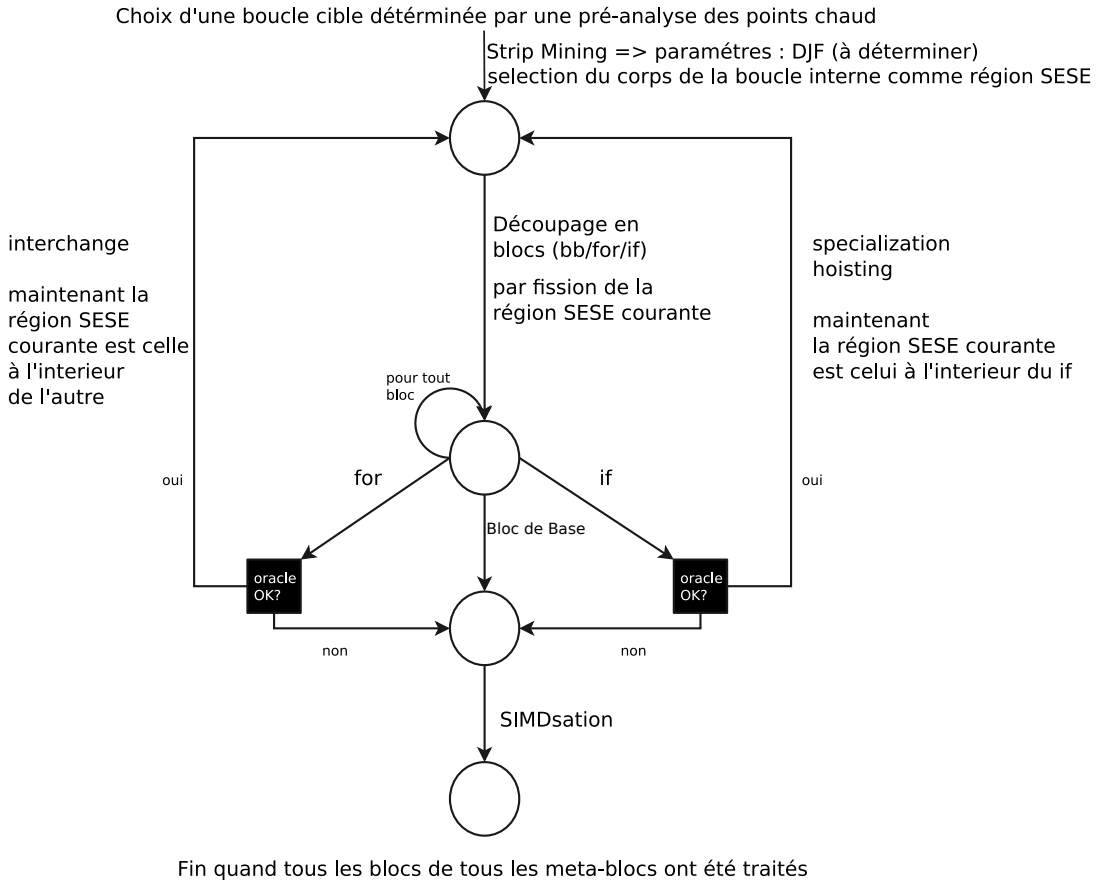


FIGURE 3.2 – Graphe orienté pour le traitement du flot de contrôle avec les oracles

Avant de commencer à définir les boites noires ou oracles, il est intéressant de mettre en lumière les besoins et l'importance de celles-ci.

3.2.3 Exemple d'impact du support du SIMD sur deux architectures

Si nous regardons un code simple contenant une boucle avec un `if` inclus, il faut retenir de la Figure 3.3 que l'accélération pour le programme tournant sur l'architecture MIC est plus élevée que sur Haswell, même en relativisant avec le maximum possible sur les machines (respectivement 8 et 4). Donc il faut prendre en compte le fait que l'architecture à un rôle décisif dans un modèle de coût surtout au vu du fait que des architectures différentes n'ont pas forcement le même jeu d'instruction ni les mêmes registres.

3.2.4 Exemple de l'impact du choix du compilateur

Un deuxième phénomène important à mettre en valeur est, cette fois-ci, de fixer le paramètre de l'architecture (plus précisément dans cet exemple Haswell) tout en faisant varier le compilateur. Dans la Figure 3.4 nous avons comparé ICC 15 et GCC 5.2.0 pour un code avec une boucle à vectoriser contant deux ifs imbriqués à

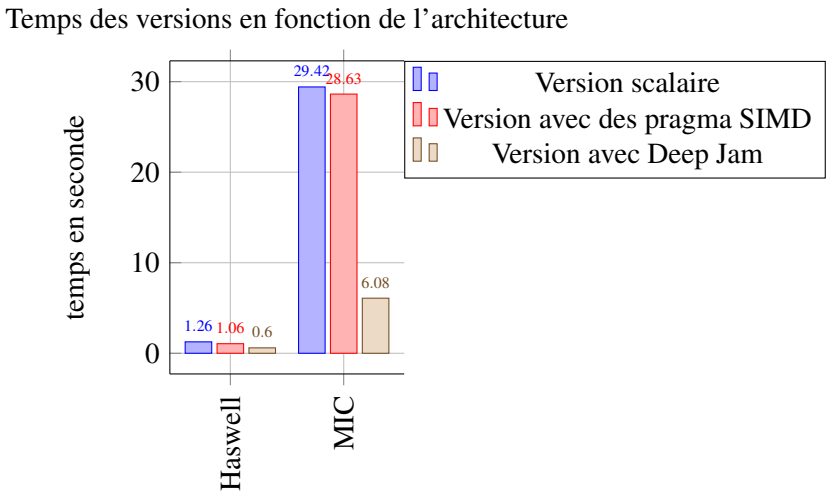


FIGURE 3.3 – Évolution du temps d'exécution en fonction de l'architecture

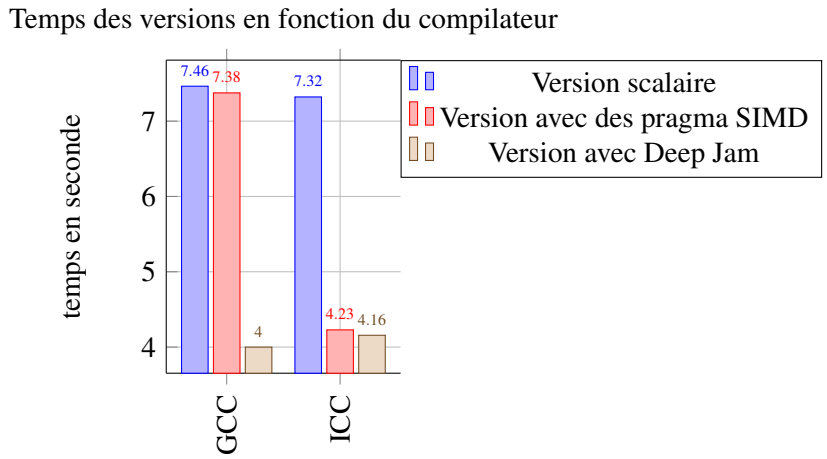


FIGURE 3.4 – Évolution du temps d'exécution en fonction du compilateur

l'intérieur. Il en ressort que la différence en temps pour les versions avec Deep Jam appliqué est très proche, en revanche l'écart du temps en scalaire est nettement plus important. Nous avons donc un compilateur beaucoup plus agressif, mais moins performant, ICC, sur Haswell. Ainsi nous devons aussi prendre en compte les différents compilateurs tout en retenant qu'il y a deux aspects : le temps absolu et l'accélération.

3.2.5 Synthèse des besoins

Parfois une vectorisation trop aggressive a un impact négatif sur les performances. Le Deep Jam peut ne pas être assez intéressant par rapport au simple ajout de la directive SIMD comme, par exemple, sur la Figure 3.4. Mais le Deep Jam peut s'avérer efficace dans le cas où le compilateur a plus de difficultés à transformer automatiquement le code. C'est de ce constat qu'il apparait important de concevoir des microbenchenmarks pour pouvoir prendre la décision d'appliquer une transformation de l'algorithme ou non. De plus, ces programmes serviront à expliciter les boites noires, ce qui nous amène au besoin de trouver un modèle de coût pour définir ces oracles.

3.3 Des microbenchmarks pour expliciter les oracles

Ces benchmarks ont été conçus pour déterminer les conditions favorables à l'application du Deep Jam étendu. Ainsi le choix d'appliquer une transformation ou non serait, à terme, possible. Nous détaillerons dans un premier temps trois codes pour dégager des heuristiques sur le flot de contrôle, nous ouvrirons notre étude ensuite sur un micro benchmark pour le flot de données.

Pour le flot de contrôle, les microbenchmarks définis sont : un for externe (sur lequel repose la vectorisation) et, à l'intérieur plusieurs combinaisons, `if`, `if for`, `if if` avec les `else` pour chacun pour partir sur la droite du graphe 3.2. Pour compléter, il y a aussi les programmes avec un for contenant soit un `for`, soit un `for if` (partant dans un premier temps de la gauche du graphe que nous n'avons pas encore exploité). Quant au flot de données, nous sommes partis du benchmark `if for`. Tous les microbenchmarks contiennent en plus une boucle `for` qui encapsule les opérations. Pour les résultats, nous allons analyser l'impact des paramètres de la simulation : le Facteur de Deep Jam (DJF) et le pourcentage du nombre de fois que le chemin choisi (celui qui va être optimisé) est pris.

3.3.1 Microbenchmark n° 1 : for if / for if else

Ici nous étudions l'ajout d'un unique niveau de flot de contrôle dans une simple boucle `for`. Ce qui nous permet d'observer le coût du Deep Jam dans un code simple (très peu irrégulier). Le pseudo-code scalaire sans `else` est visible sur le Listing 3.19 et avec sur le Listing 3.20 :

Listing 3.19 – for if scalaire

```
1 pour i : 0 → N
2     si c1
3         A
4     fin si
5 fin pour
```

Listing 3.20 – for if else scalaire

```
1 pour i : 0 → N
2     si c1
3         A
4     sinon
5         B
6     fin si
7 fin pour
```

Le pseudo-code avec le Deep Jam est illustré sur le Listing 3.21 et le Listing 3.22. Nous avons commencé par décider que la branche où l'instruction `if` est validée est celle à privilégié. Nous avons fait le couple de transformations spécialisation plus hoisting :

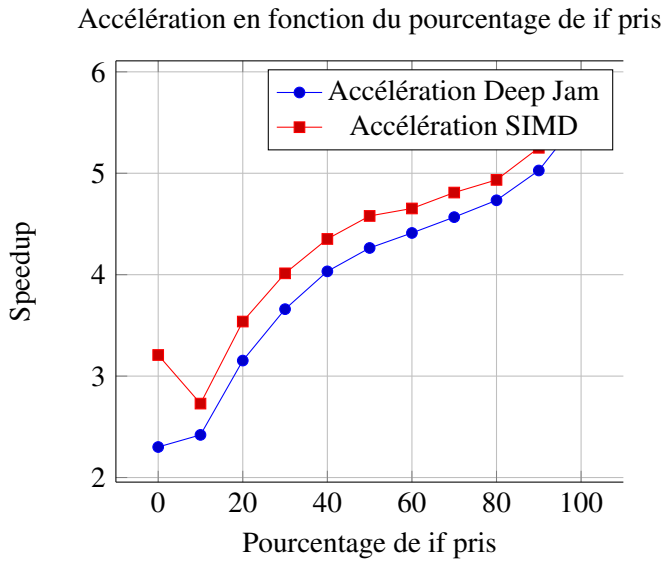


FIGURE 3.5 – Évolution du speedup fonction du pourcentage de if pris sur le benchmark « for if »

Listing 3.21 – for if avec Deep Jam

```
1  pour i : 0 → N , i+=DJF
2      pour j : i → i+DJF
3          cc ← cc ∧ c1_j
4      fin pour
5      si cc
6          pour j : i → i+DJF
7              A
8          fin pour
9      sinon
10         pour j : i → i+DJF
11             si c1
12                 A
13             fin si
14         fin pour
15     fin si
16 fin pour
```

Listing 3.22 – for if else avec Deep Jam

```
1  pour i : 0 → N , i+=DJF
2      pour j : i → i+DJF
3          cc ← cc ∧ c1_j
4      fin pour
5      si cc
6          pour j : i → i+DJF
7              A
8          fin pour
9      sinon
10         pour j : i → i+DJF
11             si c1
12                 A
13             sinon
14                 B
15             fin si
16         fin pour
17     fin si
18 fin pour
```

Sur la Figure 3.5 nous pouvons observer que pour ce benchmark quelque soit le rapport entre le nombre de fois où la condition `c` est prise et le nombre de fois où elle ne l'est pas la version avec le Deep Jam reste moins performante que celle avec juste le pragma SIMD. Sur le MIC, cela s'explique par le fait que ce microbenchmark reste très basique et donc permet au compilateur d'avoir une marche de main-d'œuvre conséquente avec le pragma SIMD. Le Deep Jam, dans ce cas, reste trop complexe.En revanche sur un processeur Haswell l'architecture un jeu d'instruction vectorielle moins complet. Le compilateur d'Intel ne peut donc pas être aussi agressif que sur le MIC.

3.3.2 Micro-benchmark n° 2 : for if if

Dans ce benchmark nous étudions l'impact du flot contrôle imbriqué dans une simple boucle. Le pseudo-code scalaire est détaillé sur le Listing 3.23 (sans `else`) et le Listing 3.24 (avec `else`) :

Listing 3.23 – for if if scalaire

```
pour i : 0 → N
    si c1
        si c2
            A
        fin si
    fin si
fin pour
```

Listing 3.24 – for if if else else scalaire

```
pour i : 0 → N
    si c1
        si c2 A
        sinon B
    sinon C
fin pour
```

Le pseudo-code avec Deep Jam est le suivant, cf. le Listing 3.25 (sans `else`) et le Listing 3.26 (avec `else`), pour cela nous avons fait une double spécialisation/hoisting pour chaque if. En effet nous commençons par faire face à un test `si` c_1 et comme le chemin privilégié est celui où c_1 et c_2 sont pris, d'après la Figure 3.2 nous réalisons une spécialisation puis un hoisting. Ensuite nous plongeons dans la condition `si` c_2 et répétons la même série de transformations :

Listing 3.25 – for if if avec Deep Jam

```
pour i : 0 → N , i+=DJF
    pour j : i → i+DJF
        cc1 ← cc1 ∧ c1_j
    fin pour
    si cc1
        pour j : i → i+DJF
            cc2 ← cc2 ∧ c2_j
        fin pour
        si cc2
            pour j : i → i+DJF
                A
            fin pour
        sinon
            pour j : i → i+DJF
                si c2
                    A
                fin si
            fin pour
    sinon
        pour j : i → i+DJF
            si c1
                si c2
                    A
                fin si
            fin si
        fin pour
    fin si
fin pour
```

Listing 3.26 – for if if else else avec Deep Jam

```
pour i : 0 → N , i+=DJF
    pour j : i → i+DJF
        cc1 ← cc1 ∧ c1_j
    fin pour
    si cc1
        pour j : i → i+DJF
            cc2 ← cc2 ∧ c2_j
        fin pour
        si cc2
            pour j : i → i+DJF
                A
            fin pour
        sinon
            pour j : i → i+DJF
                si c2
                    A
                sinon
                    B
                fin si
            fin pour
    sinon
        pour j : i → i+DJF
            si c1
                si c2
                    A
                sinon
                    B
                fin si
            sinon
                C
            fin si
        fin pour
    fin si
fin pour
```

On peut voir le temps des différentes versions sur la Figure 3.6, plus précisément nous avons une première version avec un `if` associé à son else et la seconde lorsqu'on a deux `if` imbriqués. Nous pouvons noter qu'ici avec 100% de fois passées dans le premier `if` (et dans les deux premiers `if` pour la seconde version), le Deep Jam ne rivalise pas avec la version comportant des directives SIMD. En revanche quand le pourcentage pris est de seulement 50, le Deep Jam s'avère nettement plus intéressant. Ceci est dû au fait que les paquets ne sont pas forcement plein ce qui implique un désavantage pour le compilateur qui génère un code optimisé se basant sur des masques. Un résumé des résultats sur Haswell en fonction des compilateurs est visible sur

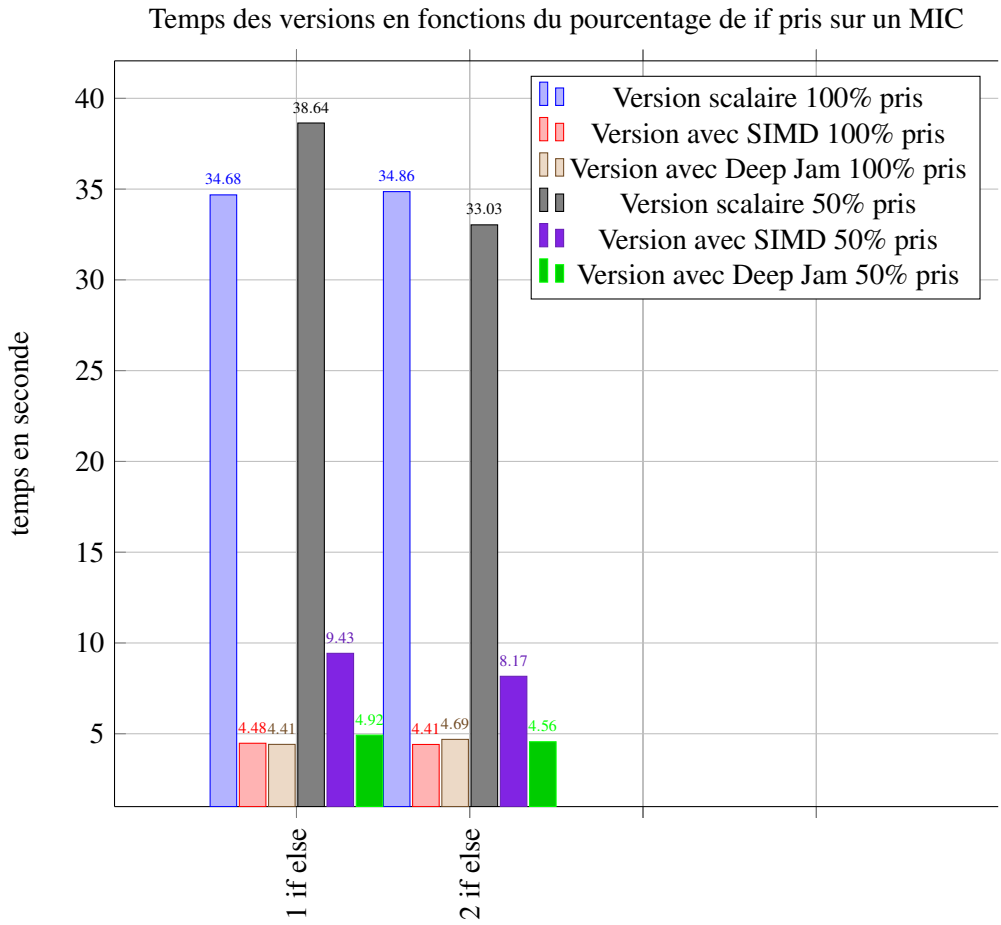


FIGURE 3.6 – Évolution du temps d'exécution en fonction du pourcentage de if pris pour chaque version du bench « for if if »

la Figure 3.7, nous pouvons observer sur cet histogramme le fait que le Deep Jam permet d'avoir un gain de temps non négligeable. De plus nous pouvons constater que l'ajout seul du pragma SIMD a pour conséquence de dégrader les performances dans certains cas. De plus, quand nous avons 100 % des conditions prises avec le Deep Jam certes le code passe par la version simplifiée à chaque fois, mais pour le pragma la tâche est aussi facilitée, car iles paquets sont pleins.

3.3.3 Micro-benchmark n° 3 : for if for / for if for else for

Nous regardons dans ce code l'ajout de boucles conditionnées (boucle dans un if lui-même dans un for) nous cherchons de nouveau à optimiser la boucle externe. Pour ce microbenchmark nous avons détaillé les différentes étapes pour arriver au code totalement transformé avec notre variante du Deep Jam, cela nous servira après dans les résultats. Le pseudo-code scalaire est visible sur Listing 3.27 (sans `else`)et le Listing 3.28 (avec `else`) :

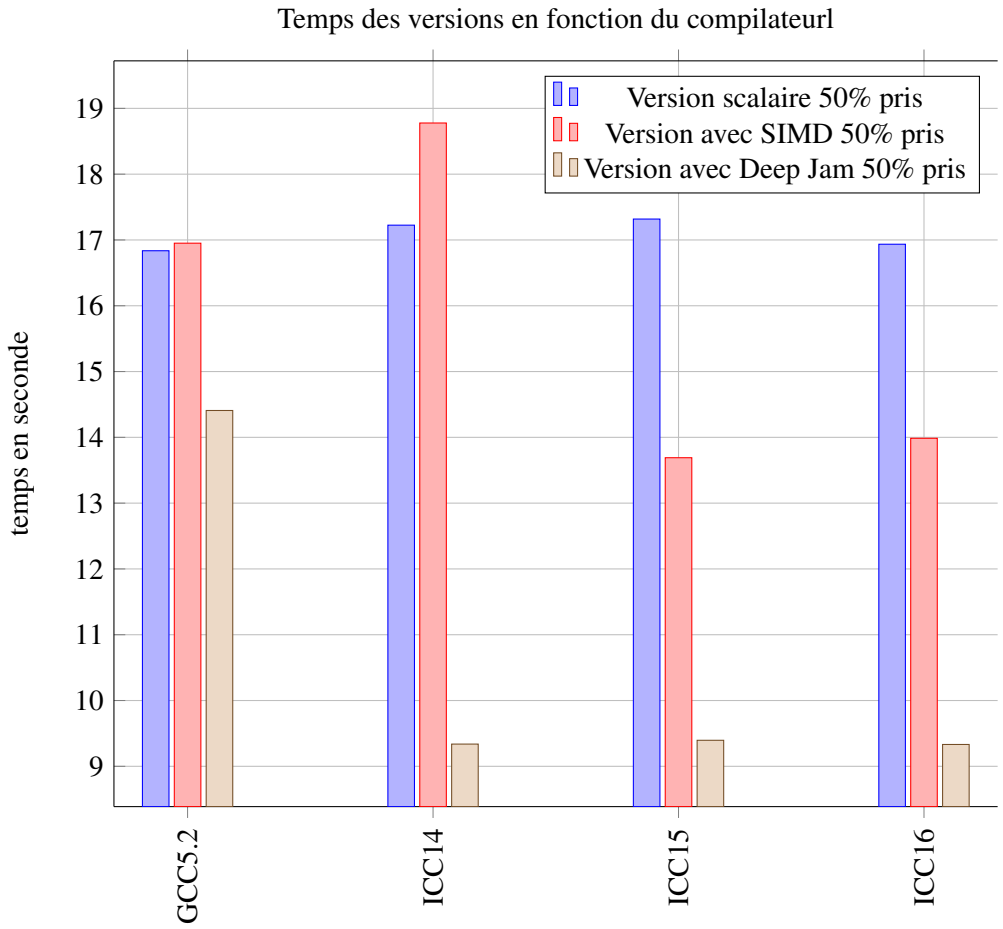


FIGURE 3.7 – Évolution du temps d'exécution en fonction du compilateur sur le benchmark « for if if »

Listing 3.27 – for if for scalaire

```
1  pour i : 0 → N
2      si c1i
3          pour j : 0 → M
4              A
5          fin pour
6      fin si
7  fin pour
```

Listing 3.28 – for if for else for scalaire

```
1  pour i : 0 → N
2      si c1i
3          pour j : 0 → M
4              A
5          fin pour
6      sinon
7          pour k : 0 → L
8              B
9          fin pour
10     fin si
11 fin pour
```

1. Pseudo-code après Strip Mining, une boucle interne a été ajoutée en faisant un stride de DJF sur la boucle externe comme nous pouvons l'observer sur le Listing 3.29 (sans `else`) et le Listing 3.30 (avec `else`) :

Listing 3.29 – for if for après Strip Mining

```
pour ii : 0 → N, ii+=DJF
    pour i : ii → ii+DJF
        si c1_i
            pour j : 0 → M
                A
            fin pour
        fin si
    fin pour
fin pour
```

Listing 3.30 – for if for else for après Strip Mining

```
pour ii : 0 → N, ii+=DJF
    pour i : ii → ii+DJF
        si c1_i
            pour j : 0 → M
                A
            fin pour
        sinon
            pour k : 0 → L
                B
            fin pour
        fin si
    fin pour
fin pour
```

2. Face au premier test nous faisons une spécialisation pour privilégier le cas où le premier if est validé, le pseudo-code après Spécialisation est disponible sur le Listing 3.31 (sans `else`) et le Listing 3.32 (avec `else`) :

Listing 3.31 – for if for après Spécialisation

```
pour ii : 0 → N, ii+=DJF
    bool cc=false
    pour j : ii → ii+DJF
        cc ← cc ∧ c1_j
    fin pour
    pour i : ii → ii+DJF
        pour j : 0 → M
            si cc
                A
            sinon si si c1_i
                A
            fin si
        fin pour
    fin pour
fin pour
```

Listing 3.32 – for if for else for après Spécialisation

```
pour ii : 0 → N, ii+=DJF
    bool cc=false
    pour j : ii → ii+DJF
        cc ← cc ∧ c1_j
    fin pour
    pour i : ii → ii+DJF
        pour j : 0 → M
            si cc
                A
            sinon si c1_i
                pour j : 0 → M
                    A
                fin pour
            sinon
                pour k : 0 → L
                    B
                fin pour
            fin si
        fin pour
    fin si
fin pour
```

3. Vu que la spécialisation a créé un invariant de boucle, nous pouvons mettre la boucle sur `i` à l'intérieur du test cf. le Listing 3.33 (sans `else`) et le Listing 3.34 (avec `else`) :

Listing 3.33 – for if for après Hoisting

```
1  pour ii : 0 → N, ii+=DJF
2      bool cc=false
3      pour j : ii → ii+DJF
4          cc ← cc ∧ c1_j
5      fin pour
6      si cc
7          pour i : ii → ii+DJF
8              pour j : 0 → M
9                  A
10             fin pour
11         fin pour
12     sinon
13         pour i : ii → ii+DJF
14             si c1_i
15                 pour j : 0 → M
16                     A
17                 fin pour
18             fin si
19         fin pour
20     fin si
21 fin pour
```

Listing 3.34 – for if for else for après Hoisting

```
1  pour ii : 0 → N, ii+=DJF
2      bool cc=false
3      pour j : ii → ii+DJF
4        cc ← cc ∧ c1_j
5      fin pour
6      si cc
7          pour i : ii → ii+DJF
8              pour j : 0 → M
9                  A
10             fin pour
11         fin pour
12     sinon
13         pour i : ii → ii+DJF
14             si c1_i
15                 pour j : 0 → M
16                     A
17                 fin pour
18             sinon
19                 pour k : 0 → L
20                     B
21                 fin pour
22             fin si
23         fin pour
24     fin si
25 fin pour
```

4. Face à la boucle sur `j` nous intervertissons les deux boucle et le pseudo-code avec Deep Jam complet (après interchange) est visible sur le Listing 3.35 (sans `else`)et le Listing 3.36 (avec `else`) :

Listing 3.35 – for if for avec Deep Jam

```
1  pour i : 0 → N i+=DJF
2      pour j : i → i+DJF
3          cc ← cc ∧ c1_j
4      fin pour
5      si cc
6          pour j : 0 → M
7             pour ii : i→i+DJF
8                 A
9             fin pour
10         fin pour
11     sinon
12         pour ii : i→i+DJF
13             si c1_i
14                 pour j : 0 → M
15                     A
16                 fin pour
17             fin si
18         fin pour
19     fin si
20 fin pour
```

Listing 3.36 – for if for else for avec Deep Jam

```
1  pour i : 0 → N i+=DJF
2       pour j : i → i+DJF
3           cc ← cc ∧ c1_j
4      fin pour
5      si cc
6          pour j : 0 → M
7             pour ii : i→i+DJF
8                 A
9             fin pour
10         fin pour
11     sinon
12         pour ii : i→i+DJF
13             si c1_i
14                 pour j : 0 → M
15                     A
16                 fin pour
17             sinon
18                 pour k : 0 → L
19                     B
20                 fin pour
21             fin si
22         fin pour
23     fin si
24 fin pour
```

Sur la Figure 3.8 nous avons le temps du benchmark « for if for » avec l'impact du DJF sur le temps d'exécution, nous pouvons constater que, de manière assez logique (vu que l'architecture cible peut travailler sur des paquets de 8 nombres réels), lorsque ce facteur vaut 8 que les performances sont les meilleures.

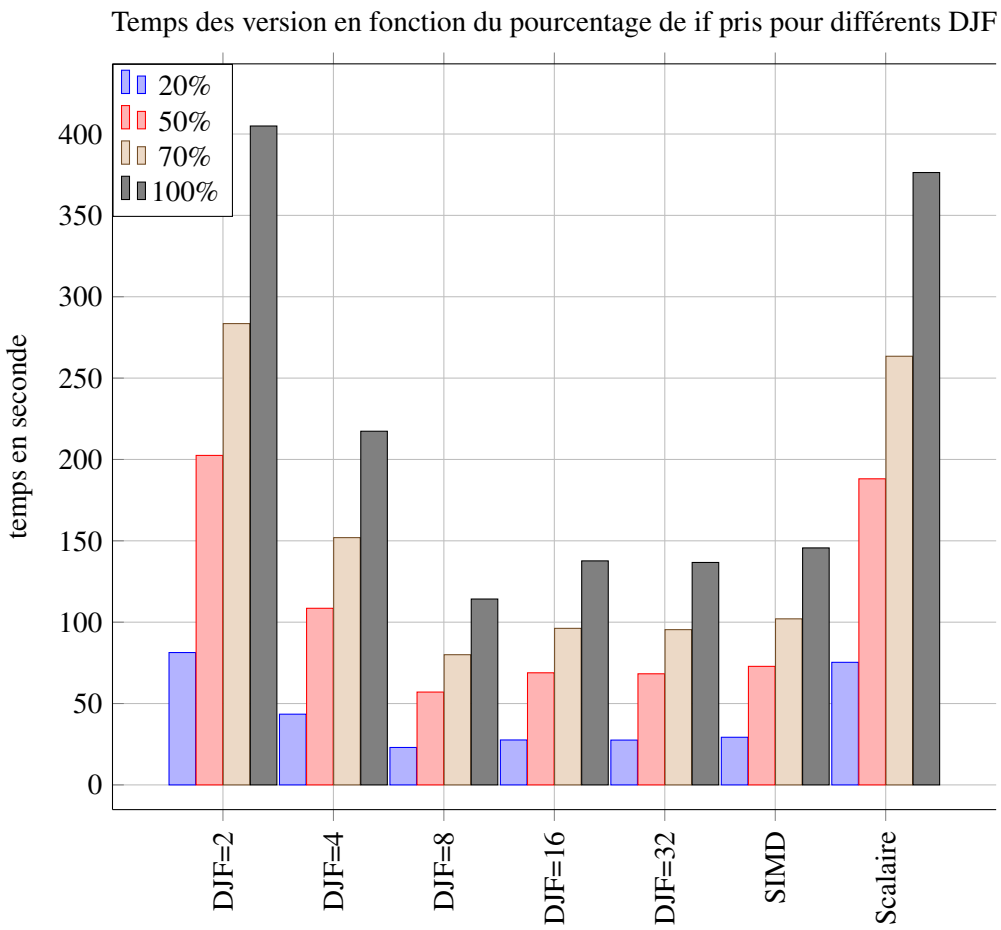


FIGURE 3.8 – Évolution du temps d'exécution en fonction de la version et du pourcentage de if pris pour le benchmark « for if for »

Quant à la Figure 3.9 elle montre l'impact des différentes étapes appliquées sur le benchmark, ce qui en ressort c'est que le pragma SIMD fait la différence même si l'interchange à un impact plus visible que les précédentes étapes.

Trop complexe pour la vectorisation automatique, le SIMD a besoin du Deep Jam pour être intéressant, car il est appliqué sur une boucle sans flot de contrôle. De plus, le DJF de 8 est le meilleur, car il est en accord avec la taille des registres vectoriels sur le Xeon Phi (qui peuvent opérer sur 8 doubles en même temps)

3.3.4 Réflexion pour le flot de données

Une nouvelle réflexion concerne, cette fois-ci, la régularité des tableaux utilisés au sein d'un *threadlet*. En effet, un accès irrégulier dans un contexte de vectorisation implique l'utilisation, si disponible, d'instructions vectorielles de `scatter` et `gather`, opérations gourmandes en temps (leur fonctionnement est visible sur la Figure 3.10, un gather va chercher des données en mémoire de manière non contiguë pour les mettre dans un vecteur et le scatter fait l'inverse). En cas d'absence de ces instructions, le surcout est encore plus important, le compilateur est alors obligé de générer du code scalaire forçant des aller-retour scalaires vectoriels.

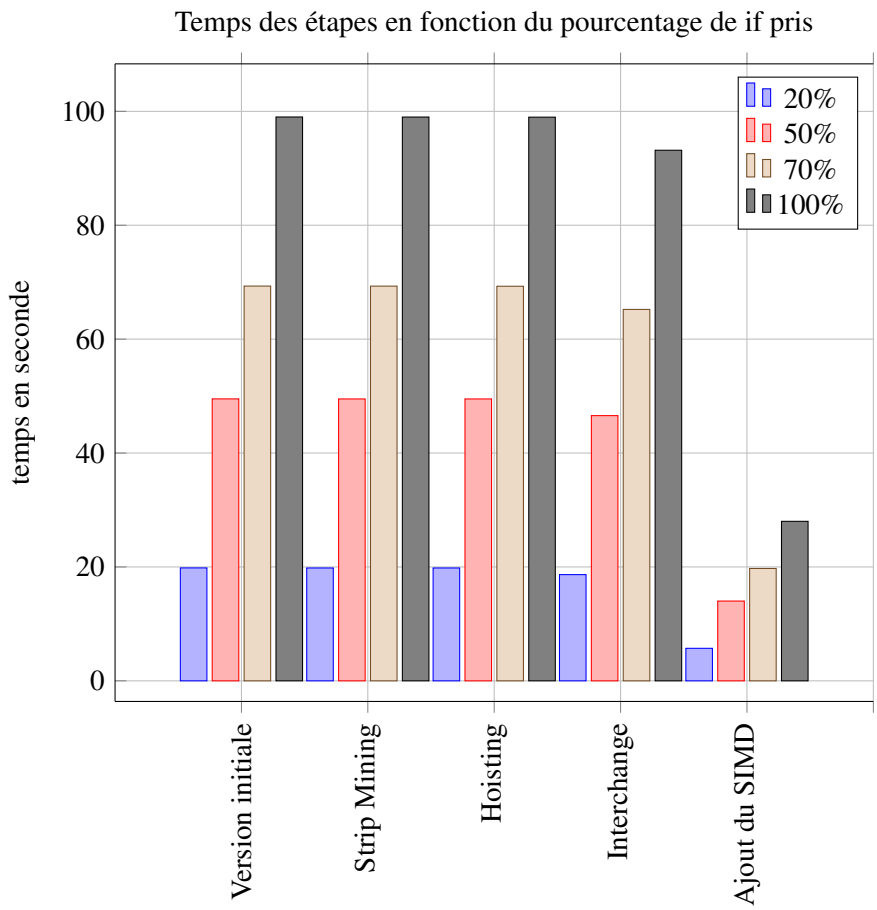


FIGURE 3.9 – Évolution du temps d'exécution en fonction du pourcentage de if pris pour chaque étape sur le benchmark « for if for »

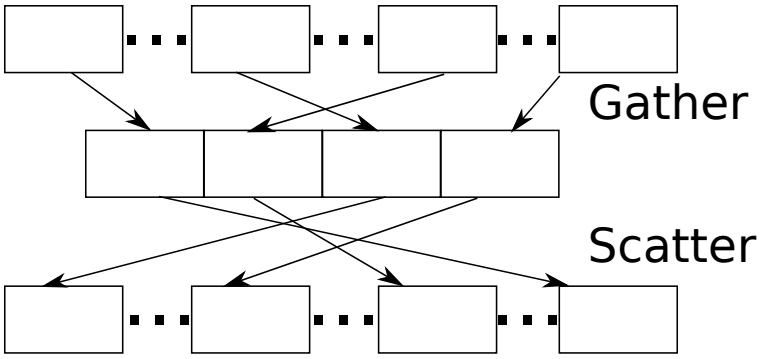


FIGURE 3.10 – Schéma du fonctionnement du scatter et du gather

Microbenchmark avec indirection, rôle et fonctionnement

Le code est le même que ceux du benchmark n° 3, visible sur le Listing 3.35 à ceci près que des accès indirects aux tableaux (du type a[b[i]]) ont été rajoutées. Cette modification permet d'apporter un flot de données irrégulier à nos tests. Nous pouvons observer sur la Figure 3.11 les résultats du benchmark : malgré l'ajout des indirections, les performance sont peu impactées.

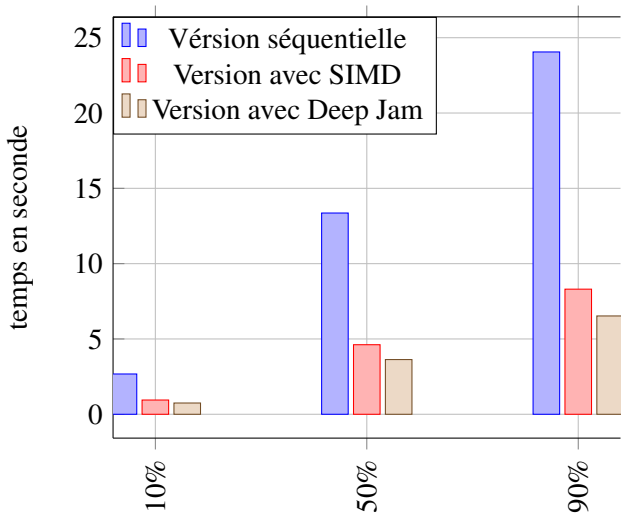


FIGURE 3.11 – Évolution du temps d'exécution en fonction du pourcentage de if pris pour chaque étape sur le benchmark « for if for » avec indirections

3.3.5 Résumé et conclusions

Ces microbenchmarks nous ont permis de mettre en lumière différentes heuristiques pour appréhender les oracles de la Figure 3.2. Même si cette liste n'est exhaustive, elle nous a permis d'avoir les conclusions suivantes :

- quand le corps de boucle est simple avec juste un niveau de flot de contrôle le compilateur s'en sort mieux lorsqu'il est sur une architecture où le jeu d'instruction propose des mécanismes simples pour gérer les branchements (masques) sinon le code optimisé par le Deep Jam permet d'améliorer les performances ;
- l'imbrication de flot de contrôle dans un code optimisé le Deep Jam est favorable si la condition externe n'est pas toujours validée sur le MIC. En revanche sur Haswell l'ajout d'un pragma SIMD (ou omp simd avec gcc) ne rivalise pas avec l'implémentation du Deep Jam ;
- avec un benchmark plus complexe, l'ajout d'un for dans un if, lorsqu'on sélectionne le DJF il apparait que, de manière optimale vaut mieux qu'il vaille la taille d'un registre ;
- toujours avec le programme "for if for" en s'attardant sur l'enchainement d'optimisation pour aboutir à la version optimisée avec notre variante du Deep Jam celle-ci n'apporte que peu chacune, c'est au moment de rajouter le pragma sur les différentes boucles que le gain se fait réellement ;
- enfin, cette fois-ci en éditant le benchmark précédent pour qu'il contienne des indirection (flot de données irrégulier) sur le MIC le Deep Jam s'avère fortement intéressant, l'architecture comprenant des instructions vectorielles de type scatter/gather.

3.4 Conclusions, limites

Nous avons désormais défini notre variante du Deep Jam, celle-ci contient deux oracles (aussi appelés boites noires) pour déterminer quand arrêter son application, utiles lorsque nous sommes face à de multiples branchements ou de profondes imbrications. C'est pour amener une première réflexion et de premières heuristiques que les microbenchmarks ont été définis. Cette étude a aussi montré des limites : lorsque nous

voudrons inférer sur le flot de contrôle, l'analyse et les transformations pourront se faire statiquement alors que pour le flot de données cela ne peut être fait que dynamiquement. Donc à défaut d'avoir pu implémenter l'algorithme dans un compilateur source à source, nous allons voir son application sur un benchmark afin de le valider « manuellement ».

Chapitre 4

Application manuelle du Deep Jam adapté sur un code de calcul

Nous avons donc désormais à notre disposition un algorithme Deep Jam adapté pour apporter du parallélisme vectoriel dans un code irrégulier. Mais celui-ci n'a été appliqué, pour le moment, que sur des petits cas tests pour déterminer quand et comment l'utiliser. C'est pour cela qu'ici nous allons l'utiliser pour vectoriser un programme plus réaliste et conséquent, cela permettant alors de valider l'algorithme et de tester plus généralement ses performances. Dans ce chapitre, nous présenterons le benchmark utilisé pour l'application de notre variante du Deep Jam. Puis nous détaillerons notre démarche pour terminer sur les résultats obtenus.

4.1 Le benchmark HydroMM

Nous avons choisi un benchmark à la fois représentatif d'un cas réel tout en restant relativement petit. Son principal intérêt est qu'il nous a permis d'étudier plus profondément le cas de la vectorisation sur du code irrégulier en déroulant manuellement notre variante du Deep Jam. En effet, son flot de contrôle est intéressant (sans être pour autant trop compliqué à étudier) avec des boucles sans dépendances inter-itérations. Cette configuration nous laisse de la marge de travail vis-à-vis des performances. De plus, ce code contient, dans certaines boucles, du flot de données irrégulier c'est-à-dire des accès indirects comme : `a[b[i]]`.

HydroMM est un code d'hydrodynamique-multi matériaux conçu par le CEA pour simuler l'écoulement de différents matériaux sur un maillage structuré en deux dimensions pour des architectures hybrides, même si ici nous nous intéresserons uniquement sur la version pour CPU. En effet, le caractère hybride signifie qu'il peut s'exécuter à la fois sur CPU et GPU. Ce programme après l'initialisation exécute une boucle où chaque itération est un pas de temps. À l'intérieur nous avons l'alternance de deux phases, comme nous pouvons le voir sur la Figure 4.1[25]. Sur ce graphique nous avons quatre mailles ayant respectivement comme coordonnées (i, j), $(i, j+1)$, $(i+1, j)$ et $(i+1, j+1)$. D'un point de vue numérique, il y a ainsi :

1. la phase lagrangienne, dans laquelle le maillage se déforme et « suit » la matière cf. la Figure 4.1a) vers b) [56][25] en appliquant le vecteur de déformation u représenté par la flèche rouge ;
2. la phase projection où, une fois les quantités de fins de phase lagrangienne calculées sur le maillage qui a suivi la matière, ces quantités sont projetées sur le maillage fixe initial cf. la Figure 4.1b) vers c) [56][25]. Les couleurs de la maille $(i+1, j+1)$ permettent de voir les parties des cellules voisines qui vont être projetées dans la cellule.

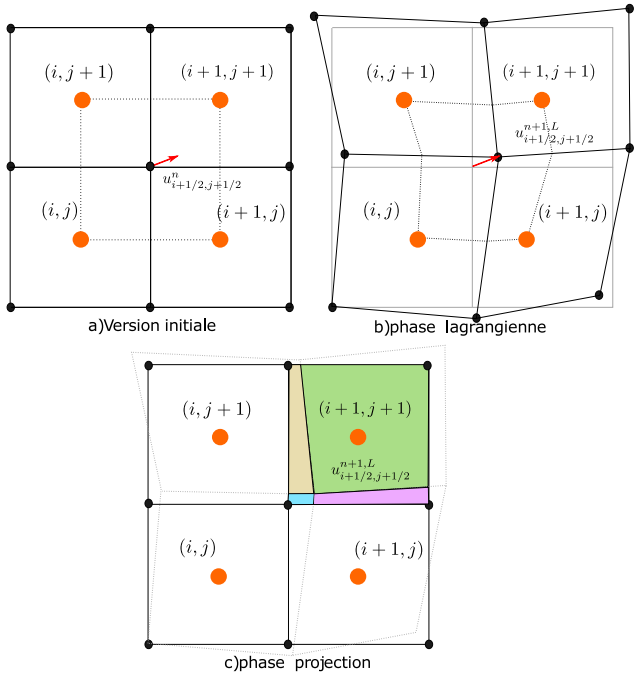


FIGURE 4.1 – Illustration des deux phases

Nous pouvons voir sur le Listing 4.1 la succession des étapes de la boucle en temps, la simulation se finissant si le nombre d'itérations a dépassé son maximum ou si le temps simulé a dépassé sa borne supérieure. La boucle commence par incrémenter le compteur d'itération pour ensuite augmenter le temps simulé de dt, cette variable est déterminée en calculant la cfl (la condition de Courant–Friedrichs–Lewy [1]). Puis les deux phases sont exécutées puis les directions sont mises à jour. Suivent ensuite deux phases optionnelles : la gestion d'une deuxième direction et la sauvegarde de l'état intermédiaire.

Listing 4.1 – Déroulement de la fonction principale de HydroMM

```
1  void hydroMultiCPU(...){
2      initialisation
3      tantque la simulation n'est pas finie
4      faire
5          incrémentation du nombre d'itérations
6          Choix du pas de temps en calculant la cfl
7          Mise a jour du temps écoule
8          Alternance des directions d'une itération à une autre
9          Phase lagrangienne
10         Phase de projection
11         Mise à jour des directions
12
13         si la simulation n'est pas monoDirectionnel
14             Phase lagrangienne (direction 2)
15             Phase de projection (direction 2)
16             Mise a jour
17         fin si
18
19         Sauvegarde de l'état intermédiaire si demande
20     fin tantque
21 }
```

Maintenant, détaillons la phase lagrangienne telle qu'elle a été codée dans HydroMM cf. le Listing 4.2. Celle-ci commence par gérer les mailles fantômes pour ensuite calculer la vitesse et la pression dans chacune des

1. http://www.stat.uchicago.edu/~lekheng/courses/302/classics/courant-friedrichs-lewy.pdf

directions. Suite à cela se déroule le coeur de la fonction en débutant par la boucle sur les blocs qui était déjà parallélisée en OpenMP. Alors, deux cas sont gérés : soit le bloc est mixte (plusieurs matériaux) soit il est pur. Dans le premier cas, nous repartitionnons la variation des quantités sur chaque cellule. Dans le second cas, la masse volumique, la vitesse et l'énergie totale sont mises à jour toujours dans chaque cellule.

Listing 4.2 – Déroulement de la fonction de la phase Lagrangienne

```
1  void phaseLagrangienne(...) {
2      Mise en place des conditions aux limites (cellules fantômes)
3      conditionsLimites(...);
4      Calcul de la vitesse et de la pression aux interfaces
5      si la  direction est dir_x
6              Vitesse dans la direction x
7              fluxInterfaces(...);
8      sinon
9              Vitesse dans la direction y
10             fluxInterfaces(...);
11     fin si
12     Mise a jour des quantité
13 #pragma omp parallel for
14     pour tous les blocs
15         si le bloc est mixte
16             calcul de la position du bloc courant
17             /*Repartition de la variation des quantités
18             sur chacun des matériaux présents dans le bloc*/
19             pour toutes le cellules
20                 repartitionVariations(...);
21             fin pour
22         sinon
23             calcul de la position du bloc courant
24             //Mise a jour de la masse volumique, de la vitesse et de l'énergie totale
25             pour toutes le cellules
26                 schemaGodunov(...);
27             fin pour
28         fin si
29     fin pour
30 }
```

Voyons la manière dont fonctionne la phase de projection dans le benchmark HydroMM. Nous pouvons voir son code schématisé sur le Listing 4.3. Avant d'itérer sur tous les blocs il y a quatre appels de fonction : (i) le calcul des volumes partiels, (ii) le calcul des conditions limites, (iii)le calcul des volumes partiels et (iv)le calcul des quantités décentrées amont. Ensuite, pour tous les matériaux, dans tous les blocs nous déterminons s'il y a soit un bloc à gauche, soit à droite tout en s'assurant que le nouveau bloc courant est dans le domaine. Si cela est bon, la projection est effectuée sur chaque cellule.

Listing 4.3 – Déroulement de la fonction de la phase Projection

```
1  void phaseProjection(...) {
2      calculProprietesMoyennesProj(...); //Calcul des volumes partiels echanges aux interfaces
3      conditionsLimites(...);
4      calculVolumesPartielsInterfaces(...);
5      calculDecentrageAmont(...); // Calcul des quantités décentrées amont
6      Projection des quantités
7
8  #pragma omp parallel for
9      pour tous les blocs
10         pour tous les matériaux
11             calcul de la position du bloc courant
12             si  le bloc est dans le domaine ou soit il y a un bloc à gauche ou soit il y en a un à
   droite
13                 pour chaque cellule
14                     projection(...);
15                 fin pour
16             fin si
17         fin pour
18     fin pour
19 }
```

Nous venons de voir comment le code fonctionne, maintenant nous allons décrire la démarche suivie pour vectoriser HydroMM avec notre transformation, le Deep Jam étendu. La première étape est la détermination d'un point chaud, puis l'application détaillée de la variante.

4.2 Vectorisation du benchmark

Avant de pouvoir appliquer notre Deep Jam, il est nécessaire de trouver au moins un point chaud. En effet, nous ne cherchons pas pour le moment à appliquer le Deep Jam sur tout le code, mais plutôt cibler une boucle gourmande en temps.

4.2.1 Première étape, les points chauds

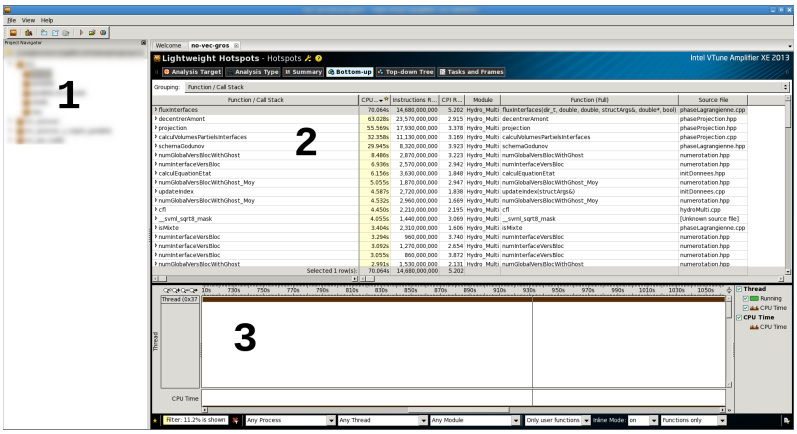
Pour trouver les points chauds, nous nous sommes servi du logiciel Intel Vtune Amplifier XE [2]. Étudier les points chauds du benchmark revient à trouver les goulots d'étranglement à l'aide d'une analyse post-mortem en regardant le « temps » passé dans chacune des fonctions.Ce n'est pas à proprement parler un temps, mais plus exactement le nombre d'instructions écoulées. Tout cela dans le but de trouver la portion de code intéressante à vectoriser. Les Figures 4.2a et 4.2b nous montre l'interface de Vtune telle que nous l'avons utilisé :

- dans le panneau de gauche, nous avons l'explorateur de projet (**1**) qui nous permet de naviguer dans les résultats et les projets ;
- en haut à droite sont exposées les informations collectées lors de l'instrumentation (**2**) ; celles-ci sont classables de différentes manières sachant que chaque ligne représente une fonction. Par exemple « Bottom-Up » trie les fonctions par temps/cycles et il est possible d'afficher la pile d'appels pour chaque item ;
- en bas à droite l'exécution de chacun des threads (la couleur verte indique que le thread ne fait rien et en marron l'inverse) (**3**), à noter qu'en dessous il y a un menu pour filtrer les résultats.

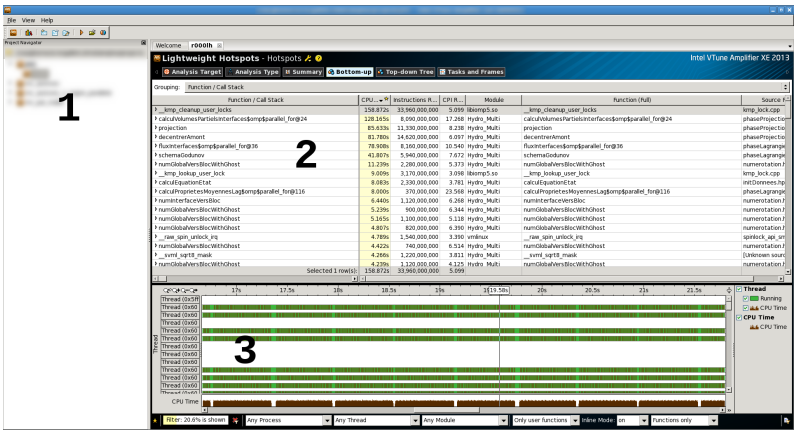
Cette étude post-mortem fournie par Intel Vtune Amplifier XE nous a permis de trouver un hotspot intéressant : la fonction `projection`, car nous pouvons voir dans la Figure 4.2b qu'elle prend le plus de temps. Ensuite, nous avons mené une étude du temps de la boucle optimisée suivant les différentes versions en calculant le grind time (temps par maille par cycle - μs/cellule/cycle).

C'est le hotspot qui a déterminé le choix la boucle à optimiser, celle qui nous offrait le plus de potentiel : une boucle dans la fonction « phaseProjection », celle appelant la fonction `projection`. Plus précisément c'est la boucle principale de cette fonction que nous avons décidé de vectoriser grâce à notre variante du Deep Jam. Les autres boucles du programme ont seulement été préfixées par un pragma SIMD, parfois strip minées en amont si elles étaient déjà parallélisées avec OpenMP lorsque c'était possible. Le squelette de la boucle à optimiser est visible sur le Listing 4.4.

2. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`



(a) Première capture d'écran avec un programme en mode séquentiel



(b) Seconde capture d'écran avec un programme en mode parallèle

FIGURE 4.2 – Captures d'écrans avec Vtune Amplifier XE

Listing 4.4 – Cœur de la boucle à optimiser

```
1  pour la cellule c : 0→NombreCellule−1
2      pos=indexBlock+index(c)
3      initialisation des index //utilisés en lecture
4
5      si α[pos] > ε faire
6          mise à jour masse volumique
7          si ρ[pos] ≤ 0  faire
8              mises à jour à l'index pos
9          sinon
10             réinitialisation de tableaux à l'index pos
11         fin si
12     fin si
13 fin pour
```

Il est important de noter qu'au vu du fait que le Deep Jam s'applique à une combinaison de branchement, il nous a fallu déterminer quel chemin serait mis en avant. Le rendu de cette analyse post-mortem est visible sur la Figure 4.3. Le chemin à optimiser est celui validant les deux « if » imbriqués, car on note très clairement la différence du nombre de cycles entre les chemins.

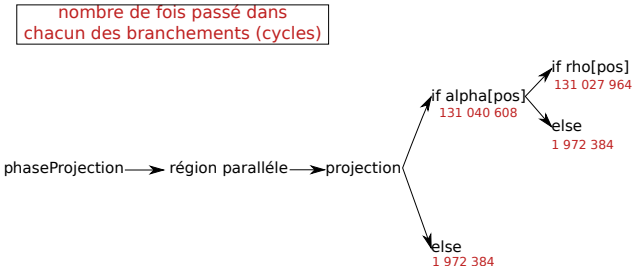


FIGURE 4.3 – Compte rendu de l'analyse Vtune pour savoir quel embranchement à favoriser

4.2.2 Flot de contrôle

Grâce à notre étude précédente, nous avons pu apporter progressivement la vectorisation via le Deep Jam. Pour cela nous avons suivi le graphe visible sur la Figure 3.2. Nous commençons par un Strip Mining avec un *stride* de valeur `DJF` (*Deep Jam Factor*. Nous chercherons par la suite la valeur idéale de `DJF` pour ce programme. Pour cela nous modifions le pas de la boucle pour le mettre à `DJF`. Ce qui fait que seulement $\lfloor \frac{NombreCellule}{DJF} \rfloor$ itérations sont traités par la boucle externe. Ce qui a pour effet de créer sune boucle interne d'itérateur `cc` qui comble les trous. Le seul changement surs le corps de la boucle est de modifié l'utilisation de `c` en `cc`. Le Listing 4.5 nous permet de voir le code transformé par rapport au code d'origine (le Listing 4.4)

Listing 4.5 – Cœur de la boucle à optimiser après Strip Mining

```
1  pour la cellule c : 0→NombreCellule-1,c+=DJF
2      pour cc : c→c+DJF
3          pos=indexBlock+index(cc)
4          initialisation des index//utilisés en lecture
5
6          si α[pos] > ε faire
7              mise à jour masse volumique
8              si ρ[pos] ≤ 0  faire
9                  mises à jour à l'index pos
10             sinon
11                 réinitialisation de tableaux à l'index pos
12             fin si
13         fin si
14     fin pour
15 fin pour
```

Désormais nous avons une boucle `cc` et d'après l'algorithme de la Figure 3.2 nous nous y plaçons dedans. Dans cette boucle, il y a un `if` sur α et des instructions (sur les lignes 3 et4) qui forment un bloc de base. Alors nous faisons une fission pour séparer le coeur de la boucle en deux régions SESE (Single Entry Single Exit). Nous traitons le bloc de base en premier qui ne nécessite pas de traitement, car nous sommes sur la branche du milieu sur la Figure 3.2 qui ne requiert pas d'oracle. Nous pouvons passer à la région suivante. À cet instant nous avons le code visible sur le Listing 4.6.

Listing 4.6 – Cœur de la boucle à optimiser après la première fission

```
1  pour la cellule c : 0→NombreCellule−1,c+=DJF
2      pour cc : c→c+DJF
3          pos=indexBlock+index(cc)
4          initialisation des index//utilisés en lecture
5      fin pour
6      pour cc : c→c+DJF
7          si α[pos] > ε faire
8              mise à jour masse volumique
9              si ρ[pos] ≤ 0  faire
10                 mises à jour à l'index pos
11             sinon
12                 réinitialisation de tableaux à l'index pos
13             fin si
14         fin si
15     fin pour
16 fin pour
```

Nous sommes maintenant face à un if sur le tableau α et nous avons vu précédemment que nous voulons favoriser le cas où celui-ci est pris. Donc l'oracle nous dirige vers l'implémentation d'une spécialisation pour prioriser les cas où tout le test α[pos]>ε est valide pour une itération de la boucle sur les cellules. Nous obtenons une nouvelle boucle pour le précalcule de la condition. Nous rajoutons aussi un nouveau test « si tous les α OK faire » et son `else`. Ainsi nous avons désormais le code visible sur le Listing 4.7.

Listing 4.7 – Cœur de la boucle après la première spécialisation

```
1  pour la cellule c : 0→NombreCellule−1,c+=DJF
2      pour cc : 0 → DJF−1
3          initialisation des index_cc//utilisés en lecture et sauvegarde du test de α de c à c+DJF
4      fin pour
5      pour j : c → c+DJF−1
6          si tous les α OK faire
7              calcul masse volumique à la position indexBlock+index(j)
8              si ρ[pos] ≤ 0  faire
9                  mises à jour à l'index indexBlock+index(j)
10             sinon
11                 réinitialisation de tableaux à l'index indexBlock+index(j)
12             fin si
13         sinon si α[pos] > ε faire
14             si ρ[pos] ≤ 0  faire
15                 mises à jour à l'index indexBlock+index(j)
16             sinon
17                 réinitialisation de tableaux à l'index indexBlock+index(j)
18             fin si
19         fin si
20     fin pour
21 fin pour
```

Avant de rentrer dans un bloc, l'algorithme nous impose de profiter du fait que le test « si tous les α OK faire » ne dépend pas de l'itérateur j (la ligne 5 du Listing 4.7) pour hoisté le test. En effet, nous sommes sur l'arc à droite de la Figure 3.2. Cela a pour conséquence du faire rentrer la boucle sur j dans les tests. Le corps du `else` restera désormais figé. Le Listing 4.8 montre le résultat de la transformation. Nous nous plaçons dans la boucle commençant à la ligne 6 sur le Listing 4.8.

Listing 4.8 – Cœur de la boucle après le premier hoisting

```
pour la cellule c : 0→NombreCellule−1,c+=DJF
    pour cc : 0 → DJF−1
        initialisation des index_cc //utilisés en lecture  et sauvegarde du test de α de c à c+DJF
    fin pour
    si tous les α OK faire
        pour j : c → c+DJF−1
            calcul masse volumique à la position indexBlock+index(j)
            si ρ[pos] ≤ 0  faire
                mises à jour à l'index indexBlock+index(j)
            sinon
                réinitialisation de tableaux à l'index indexBlock+index(j)
            fin si
        fin pour
    sinon
        pour cc : c→c+DJF
            pos=indexBlock+index(cc)
            si α[pos] > ε faire
                mise à jour masse volumique
                si ρ[pos] ≤ 0  faire
                    mises à jour à l'index pos
                sinon
                    réinitialisation de tableaux à l'index pos
                fin si
            fin si
        fin pour
    fin si
fin pour
```

Dans cette boucle, nous avons un calcul et un si/sinon donc potentiellement un bloc de base et un bloc avec du flot de contrôle. Nous sommes désormais sur la branche du milieu sur la Figure 3.2 avant de choisir quelle branche utiliser. Donc pour continuer à nous enfoncer nous réalisons une nouvelle fission et nous obtenons deux régions SESE tel que nous pouvons le voir sur la Listing 4.9.

Listing 4.9 – Cœur de la boucle après la seconde fission

```
pour la cellule c : 0→NombreCellule−1,c+=DJF
    pour cc : 0 → DJF−1
        initialisation des index_cc //utilisés en lecture  et sauvegarde du test de α de c à c+DJF
    fin pour
    si tous les α OK faire
        pour j : c → c+DJF−1
            calcul masse volumique à la position indexBlock+index(j)
        fin pour
        pour j : c → c+DJF−1
            si ρ[pos] ≤ 0  faire
                mises à jour à l'index indexBlock+index(j)
            sinon
                réinitialisation de tableaux à l'index indexBlock+index(j)
            fin si
        fin pour
    sinon
        pour cc : c→c+DJF
            pos=indexBlock+index(cc)
            si α[pos] > ε faire
                mise à jour masse volumique
                si ρ[pos] ≤ 0  faire
                    mises à jour à l'index pos
                sinon
                    réinitialisation de tableaux à l'index pos
                fin si
            fin si
        fin pour
    fin si
fin pour
```

Comme avant, nous commençons par sélectionner le bloc de base qui ne recevra aucun traitement particulier. Passons ensuite au bloc avec du flot de contrôle, l'étude précédente nous préconise de favoriser le contenu du si $\rho[pos] \leq 0$ à celui du sinon. Donc nous insérons une boucle pour détecter les cas où les conditions sont validées pour une itération de la boucle sur les cellules, c'est celle commençante à la 9 du Listing 4.10. Comme lors de la première spécialisation nous fabriquons un test pour créer le chemin où toutes les conditions vis-à-vis de ρ sont OK. Nous avons donc le test si tous les ρ OK comme nous pouvons le voir sur le Listing 4.10.

Listing 4.10 – Cœur de la boucle après la seconde spécialisation

```
1  pour la cellule c : 0→NombreCellule−1,c+=DJF
2      pour cc : 0 → DJF−1
3          initialisation des index_cc //utilisés en lecture  et sauvegarde du test de α de c à c+DJF
4      fin pour
5      si tous les α OK faire
6          pour j : c → c+DJF−1
7              calcul masse volumique à la position indexBlock+index(j)
8          fin pour
9          pour j : c → c+DJF−1
10             vérification si tous les éléments de ρ valident le test
11         fin pour
12         pour j : c → c+DJF−1
13             si tous les ρ OK  faire
14                 mises à jour à l'index indexBlock+index(j)
15             sinon si ρ[pos] ≤ 0
16                 mise à jour à l'index indexBlock+index(j)
17             sinon
18                 réinitialisation de tableaux à l'index indexBlock+index(j)
19             fin si
20         fin pour
21     sinon
22         pour cc : c→c+DJF
23             pos=indexBlock+index(cc)
24             si α[pos] > ε faire
25                 mise à jour masse volumique
26                 si ρ[pos] ≤ 0  faire
27                     mises à jour à l'index pos
28                 sinon
29                     réinitialisation de tableaux à l'index pos
30                 fin si
31             fin si
32         fin pour
33     fin si
34 fin pour
```

Le test si tous les ρ OK ne dépendant pas de j nous pouvons faire rentrer la boucle j (ligne 12 du Listing 4.10) dans le cette condition. Le reste restant figé. Comme nous pouvons plus nous enfoncer dans la fonction, nous avons obtenu la version avec la variante du Deep Jam appliquée.

Listing 4.11 – Cœur de la boucle après Deep Jam

```
pour la cellule c : 0→NombreCellule−1,c+=DJF
    pour cc : 0 → DJF−1
        initialisation des index_cc //utilisés en lecture  et sauvegarde du test de α de c à c+DJF
    fin pour
    si tous les α OK faire
        pour j : c → c+DJF−1
            calcul masse volumique à la position indexBlock+index(j)
        fin pour
        pour j : c → c+DJF−1

            vérification si tous les éléments de ρ valide le test
        fin pour
        si tout les ρ OK
            pour j c : → c+DJF−1
                mises à jour à l'index indexBlock+index(j)
            fin pour
        sinon
            pour j c : → c+DJF−1
                si ρ[pos] ≤ 0  faire
                    mises à jour à l'index indexBlock+index(j)
                sinon
                    réinitialisation de tableaux à l'index indexBlock+index(j)
                fin si
            fin pour
        fin si
    sinon
        pour cc : c→c+DJF
            pos=indexBlock+index(cc)
            si α[pos] > ε faire
                mise à jour masse volumique
                si ρ[pos] ≤ 0  faire
                    mises à jour à l'index pos
                sinon
                    réinitialisation de tableaux à l'index pos
                fin si
            fin si
        fin pour
    fin si
fin pour
```

Après toutes les implémentations, nous avons obtenu les versions suivantes pour le flot de contrôle :

1. initiale scalaire, c'est-à-dire sans aucune vectorisation, même la vectorisation automatique est désactivée grâce a l'argument « no-vec » passé au compilateur ;
2. initiale avec le `pragma SIMD` sur les boucles à optimiser ;
3. avec la variante du Deep Jam codée avec le pragma SIMD sur les autres boucles que nous pouvons voir le Listing 4.11 ;

Nous allons voir les résultats de ces versions dans la section 4.3. Mais avant détaillons la manière utilisée pour traiter le flot de données irrégulier présent dans HydroMM.

4.2.3 Une première régularisation du flot de données

Dans la boucle ciblée, nous avons différents tableaux utilisés en lecture dont trois générant des indirections. Nous pouvons voir sur le code schématisé sur le Listing 4.12 qu'il a très peu été changé avec le test de contiguïté dans un premier temps, nous verrons par la suite si au moins l'une des indirections est linéarisable sans test (c'est-à-dire toujours contiguë).

Listing 4.12 – Cœur de la boucle optimisée avec la gestion d'une inderection avec un facteur de Deep Jam DJF

```
pour la cellule c : 0→NombreCellule−1,c=c+DJF
    pos=indexBlock+index(c)
    pour i 0 → DJF−1
        détermination de la contiguïté d'un des tableaux
        stockage de l'index dans un tableau
        pour le cas ou il n'y aurais pas de contiguïté
    fin pour

    si contigue //l'accès se fait sans indirections
        pour i 0 → DJF−1
            initialisation des index_i//utilisés en lecture sans l'indirection
        fin pour
        si α[index_1 ... index_DJF] > {ε,...,ε} faire
            pour j 0 : → DJF−1
                calcul masse volumique
            fin pour
            si ρ[index_1 ... index_DJF] ≤ {0,...,0} faire
                pour j 0 → DJF−1
                    mises à jour à l'index pos+j
                fin pour
            sinon
                pour j 0 → DJF−1
                    si ρ[pos+j] ≤ 0 faire
                        mises à jour à l'index pos
                    sinon
                        réinitialisation de tableaux à l'index pos
                    fin si
                fin pour
            fin si
         sinon
            pour j 0 → DJF−1
                 si α[pos+j] > ε faire
                        mise à jour de la masse volumique
                        si ρ[pos+j] ≤ 0 faire
                            mises à jour à l'index pos
                        sinon
                            réinitialisation de tableaux à l'index pos
                      fin si
                  fin si
            fin pour
         fin si
    sinon si ce n'est pas contigue
        faire comme dans le cas ci−dessus sauf que l'on garde les indirections
    fin si
fin pour
```

Pour le flot de données, nous avons construit les 3 versions suivantes :

1. avec la variante du Deep Jam plus une des deux indirections remplacées manuellement après une étude statistique ;
2. avec la variante du Deep Jam plus les trois indirections remplacées à la main (mais rendant le code faux) ;
3. avec la variante du Deep Jam plus les trois indirections remplacées dont une conditionnée par un test non vectorisé et cette fois-ci le code est juste, nous verrons dans les résultats pourquoi un test est

resté ;

Ensuite nous allons voir et analyser les résultats de toutes les versions décrites précédemment tant pour le flot de contrôle que le flot données.

4.3 Résultats

Toutes ces modifications ont eu un impact sur les performances de HydroMM, voyons plus en détail cela en commençant par l'impact sur la vectorisation automatique. Ensuite, nous verrons l'effet des optimisations sur le flot de contrôle. Enfin, nous discuterons des performances des optimisations sur le flot de données.

4.3.1 Vectorisation automatique par le compilateur

Afin d'étudier le potentiel de vectorisation du compilateur, nous avons mesuré l'impact de l'activation de la vectorisation automatique du compilateur d'Intel sans modification du code source par rapport à la version scalaire initiale. Le résultat obtenu est visible sur le Tableau 4.1 pour l'architecture MIC avec le compilateur ICC version 15. Nous pouvons constater que le gain reste minime : l'accélération dépassant à peine un. Ceci laissant de la marge à de plus amples techniques pour apporter du parallélisme vectoriel. En effet le modèle de coût implémenté par Intel demande du code très formaté [18]. Et sachant que les boucles ont un comportement complexe (irrégulier), le compilateur perd en efficacité.

TABLE 4.1 – Impact de la vectorisation automatique

Version	Temps total (s)	Accélération
Version d'origine	918.068	1
Version avec la vectorisation automatique	864.288	1,062

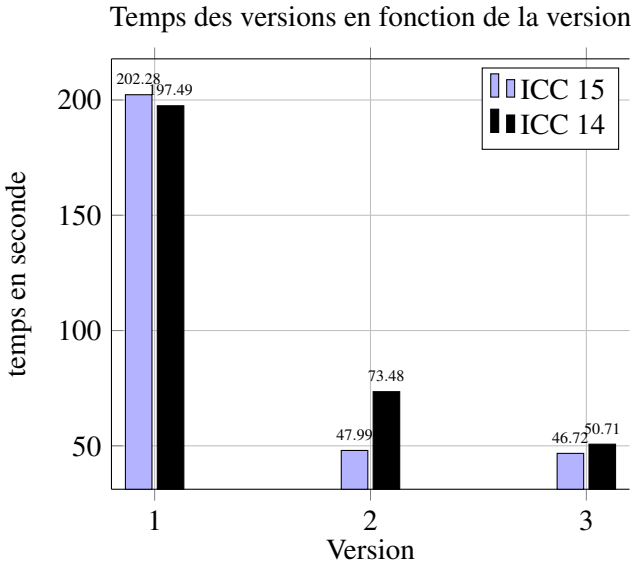
Nous avons ensuite créé la version avec l'ajout des pragma SIMD dans les des nids de boucles après avoir réalisé un Strip Mining pour conserver le parallélisme de threads utilisant OpenMP.

4.3.2 SIMD et flot de contrôle

Donc, après avoir implémenté les différentes versions rappelées à droite du graphe nous avons les performances visibles sur la Figure 4.4a avec la comparaison entre deux versions du compilateur d'Intel (14 et 15) sur un processeur KNC.

La Figure 4.5 présente le temps pris par les différentes parties du code sur la version d'origine (en rouge) et sur la version après application du Deep Jam (en bleu). On constate alors que la région parallèle de la fonction `phaseProjection` a été accélérée d'une facteur 6 sur une architecture contenant des unités vectorielles de largeur 8. Ici les autres boucles n'ont pas été optimisées et nous sommes restés en séquentiel. Donc l'application de cet algorithme sur un cas test concret a montré que, sur le flot de contrôle irrégulier, il offre un gain intéressant.

Nous pouvons ensuite étudier l'impact du facteur du Deep Jam sur ce benchmark. Comme cela est visible sur les 4.6a et 4.6b, tant sur une architecture Xeon Phi que sur un processeur Xeon un facteur de taille 32 est le plus intéressant. Cette expérience est différente des résultats du chapitre précédent. Cela nous pousse à



(a) Évolution du temps d'exécution en fonction de la version

Pour rappel les versions sont les suivantes :

1. initiale scalaire, c'est-à-dire sans aucune vectorisation, même la vectorisation automatique est désactivée grâce a l'argument « no-vec » passé au compilateur ;
2. initiale avec le `pragma SIMD` sur les boucles à optimiser ;
3. avec la variante du Deep Jam codée avec le pragma SIMD sur les autres boucles ;

(b) Les versions

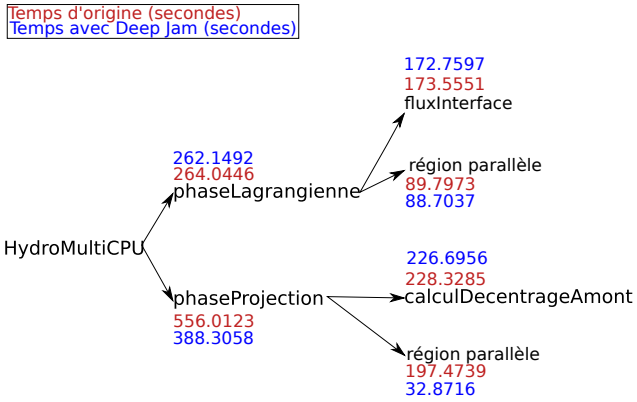
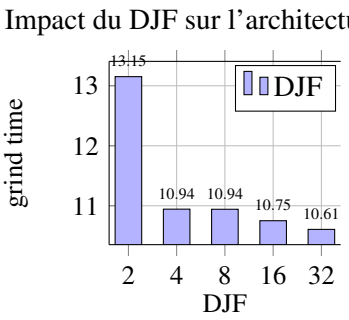
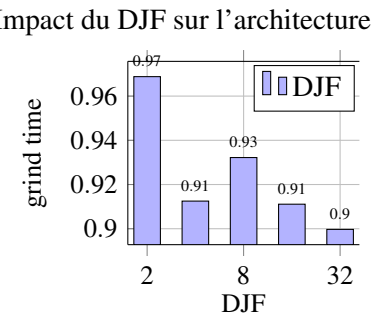


FIGURE 4.5 – Arbre des temps de fonctions après application du Deep Jam

conclure que l'optimum reste dépendant du programme et pas seulement de l'architecture. Ces résultats ont été créés en utilisant exclusivement ICC 15.





(a) Évolution du grind time en fonction du DJF sur l'architecture MIC

(b) Évolution du grind time en fonction du DJF sur l'architecture Westmere

4.3.3 Résultats pour le flot de données

On peut voir dans les Figures 4.7, 4.8 et 4.9 l'évolution des index utilisés dans la fonction pour 4 itérations. Nous pouvons observer que le premier et le troisième sont contigu par pas de temps, mais pas pour le deuxième. Cela signifie que le deuxième ne pourra pas être directement régularisé, car il est nécessaire de laisser le test de contiguïté, contrairement aux autres où cette étude statique nous permet de précalculer l'indirection.

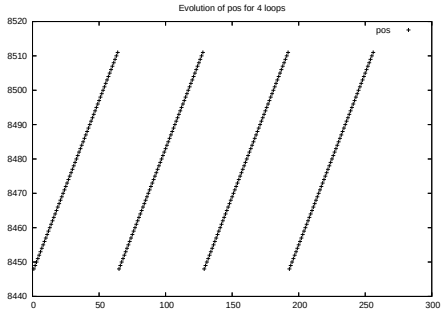


FIGURE 4.7 – Évolution du premier index

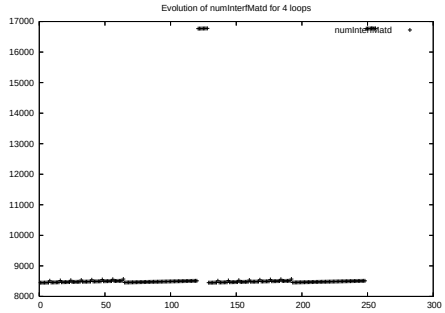


FIGURE 4.8 – Évolution du second index

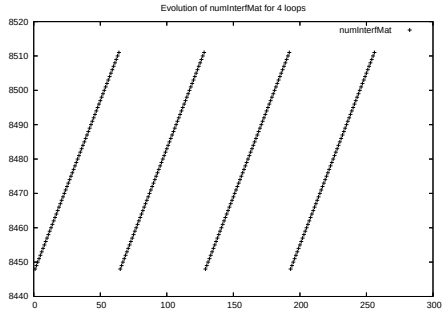


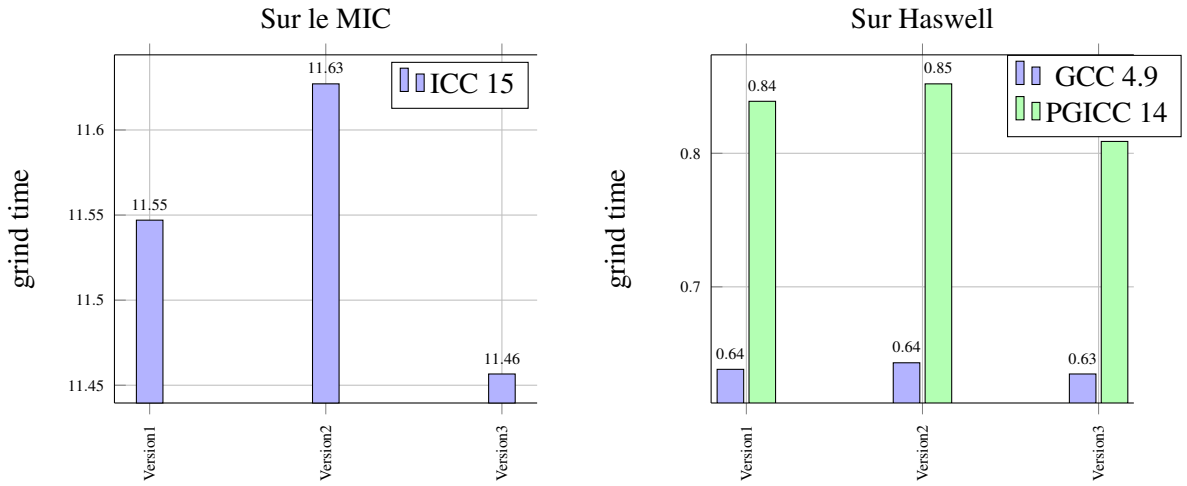
FIGURE 4.9 – Évolution du troisième index

Nous pouvons sur la Figure 4.10a et la Figure 4.10b l'impact des différentes versions sur les performances, elles sont au nombre de trois :

- la Version 1 correspond à la suppression d'une indirection sans utiliser de tests de contiguïté ;
- la Version 2 est la précédente à laquelle a été ajouté le test ;
- la dernière est la version optimale ou toutes les indirections ont été régularisées aucun sans test, donc elle est fausse au niveau des résultats numériques (l'intérêt est qu'elle sert de borne pour les performances)

Sur la Figure 4.10a nous voyons les résultats en utilisant le compilateur ICC sur l'architecture MIC. En revanche la Figure 4.10b montre les grind time des versions avec GCC 4.9 et PGICC sur l'architecture Haswell.

L'analyse de ces résultats nous permette de conclure qu'en effet la version 3 (fausse) est la meilleure et que le surcoût du test sur la version 2 est visible, et cela quel que soit l'architecture et le compilateur.



(a) Évolution du grind time en fonction des version sur le MIC

(b) Évolution du grind time en fonction des versions pour l'architecture Haswell

FIGURE 4.10 – Grind time en fonction de l'architecture

4.4 Synthèse

Pour conclure, nous avons pu vectoriser le benchmark HydroMM à l'aide de notre variante du Deep Jam, mais aussi avec des pragmas ou encore en laissant le compilateur vectoriser automatiquement. Et cela majoritairement avec ICC, mais aussi GCC ou encore PGCC. De plus, le code nous a offert l'opportunité de travailler sur le flot de données et le flot de contrôle. Donc, avec ce programme nous avons pu globalement appliquer et valider notre algorithme. Il a permis d'obtenir des gains intéressants en temps d'exécution mesurés sur les architectures Intel Xeon Phi et Intel Xeon Haswell. Aussi, au-delà de l'aspect du codage des différentes versions, nous avons obtenu des résultats probants et presque toujours cohérents avec le chapitre précédent, à une exception de la recherche de la valeur DJF optimale. Dans dernier cas, ce paramètre est dépendant du programme.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Même si le premier ordinateur dédié au calcul fut l'Eniac mis en place en 1943 l'origine des supercalculateurs trouve sa source avec le CCDC 6600, développé et vendu en 1964, pour une puissance oscillant entre 512 et 1024 KFlops. Une première révolution dans le domaine du HPC fut les machines vectorielles (de l'entreprise Cray) apparues en 1976, leur innovation se situe sur le fait de pouvoir réaliser une même opération sur un flottant sur un grand nombre de données simultanément. La seconde révolution se fit en 1980 avec l'émergence du premier cluster, un cluster étant composé de machines standards reliées entres-elles. Les avantages étant le coût à la baisse et une maintenance plus aisée. Ceci ayant entrainé une réduction des architectures utilisée pour aboutir maintenant sur une uniformisation avec une très grande majorité de clusters utilisant l'architecture x86_64 (l'architecture des PC de bureau et des servers).

Depuis la machine Eniac nous observons une augmentation exponentielle et continue des performances du à divers paramètres au cours dus temps. Celle-ci a commencé par l'accroissement des fréquences des CPU. Ensuite face à une limite en termes de consommation de ressources et d'augmentation de la chaleur émise par les processeurs (les deux étant liés), les fondeurs ont créé de nouvelles puces à plusieurs cœurs, en commençant par 2 avec l'IBM Power4 en 2001 (cadencé à 1.1 GHz). Les processeurs multi cœur se sont petit à petit imposés dans les années 2000. Mais cela a commencé à atteindre des limites dans les années qui ont suivis à cause des problèmes liés à une surconsommation. C'est ainsi que nouvelles puces ont vu le jour avec comme particularité d'avoir de nombreux cœurs légers (60 pour Intel, quelques milliers pour Nvidia). Une autre piste pour continuer d'accroître les performances commence il y a une vingtaine d'années avec la mise à disposition de registres dédiés au SIMD ont été conçus. Grâce à ces registres, il est possible d'effectuer une opération sur plusieurs données simultanément, c'est une nouvelle forme de vectorisation (le SLP). Un exemple type que nous avons longuement étudié est le Xeon Phi proposé par Intel. Pour exploiter le parallélisme vectoriel, différentes techniques existent : (i) la vectorisation automatique par le compilateur, (ii) l'ajout de directives au niveau et (iii) en utilisant des fonctions bas-niveaux, les intrinsics.

Nous avons pu conclure que sur les codes irréguliers la vectorisation est complexe, d'une part les directives ont un domaine d'application limité et d'autre part les fonctions intrinsèques ne sont pas envisageables sur un code de production, car demandant trop de refactorisation. Ainsi émergea le besoin d'extraire la régularité de ces programmes en modifiant le code soit manuellement soit automatiquement. En effet, dans la littérature différentes techniques existantes ont été étudiées . En premier lieu, il est possible avec certaines bibliothèques de modifier le code pour faire de la vectorisation explicite, le souci est le besoin de transformer le code à la

main. Selon une autre approche, il y a des façons pour apporter automatiquement le parallélisme vectoriel à une boucle dans un premier temps puis à un nid de boucle (parfait ou non) par la suite. Un autre paramètre important est la présence de flot de contrôle et son éventuelle imbrication. Mais aucune de ces diverses techniques n' est totalement adaptée à notre problème. Il faut noter que nous avons observé un manque dans la littérature correspondant à ce que nous cherchons à obtenir. Nous sommes donc partis d'un algorithme existant qui se rapprocher le plus de notre objectif, le Deep Jam. Celui-ci permet d'extraire le parallélisme à un grain fin et supporte la récursivité du code, bien que non conçu au départ pour la vectorisation.

C'est pour cette raison que nous avons adapté cet algorithme pour aboutir à une nouvelle variante qui s'appuie sur des oracles pour décider de certaines transformations à appliquer. En effet, ils servent à choisir un embranchement pour traiter ou non la région SESE (Single Entry Single Exit) courante quand ce n'est pas un bloc de base. Les régions concernées correspondent à deux types de blocs : ceux avec du flot de contrôle et ceux avec une boucle `for`. Après cela nous avons cherché et trouvé des premières heuristiques pour expliciter ces oracles à l'aide de microbenchmarks montrant des cas représentatifs. Par exemple, si le code de la boucle est très simple sans imbrication de flot de contrôle, il n'est pas probant d'appliquer le Deep Jam si nous sommes sur architecture telles que Haswell ou Sandybridge par rapport à l'architecture MIC. Nous avons aussi observé avec ces microbenchmarks le stride de la boucle à vectoriser (ce que nous avons nommé le DJF) est dépendent de l'architecture cible.

L'ensemble des heuristiques trouvées ont permis de vectoriser un code d'hydrodynamique multi-matériaux en 2 dimensions nommé HydroMM. Sur ce programme, nous avons déroulé à la main notre algorithme sur une boucle gourmande en temps, celle-ci a été trouvée grâce à une analyse des points chauds. L'avantage de cette boucle est qu'elle contient de flot de contrôle imbriqué et du flot de donnée irrégulier. Pour le flot de contrôle, il a fallu tout d'abord choisir un chemin à privilégier, car nous ne pouvons pas optimiser toutes les combinaisons de `if/else`. L'implémentation de la variante du Deep Jam a permis un gain visible de performance par rapport au simple ajout d'un `#pragma SIMD`. Nous avons fait varier aussi le compilateur : ICC GCC et PGCC, et l'architecture : MIC, Westmere et Sandybridge. Vis-à-vis du flot de données nous avions des tableaux avec des indirections et à partir d'une analyse statistique grâce à une instrumentation du code pour visualiser la contiguïté par pas de temps. Ceci nous a permis de voir qu'un seul ne l'était pas, les autres ont pu être « régularisés » (suppression de l'indirection).

5.2 Travaux futurs

Pour concure ce document, nous évoquons d'abord les perspectives à court terme pour ouvrir sur les pistes de travaux à plus longue échéance.

5.2.1 Court terme

La première tâche à réaliser sera d'expliciter plus précisément les oracles et cherchant plus d'heuristiques telles que la profondeur maximum des boucles ou des conditions imbriquées pour pour écarter les pistes où les transformations serait trop lourdes.

Le support des boucles non parfaites n'est pas, pour le moment, explicitement validé. Mais même si ce type de structure de code n'a pas été évaluée avec des microbenchmarks, notre algorithme doit être en mesure de le gérer. Un exemple schématisé d'une boucle scalaire non parfaite à vectoriser est visible sur le Listing 5.1.

Listing 5.1 – Boucle non parfaite à optimiser par le Deep Jam avec une boucle non parfaite

```
1 pour i : 0→N
2     A_i
3     si c_{1_i}
4     faire
5         B_i
6     fin si
7     D_i
8 fin pour
```

Encore vis à vis des microbenchsarks il nous manque pour cloturer l'étude les cas cas qui commencent en partant vers la branche for de l'algorithme (voir la Figure 3.2). C'est à dire des codes tels que ceux visibles sur les Figures 5.2 et 5.3.

Listing 5.2 – Boucle à optimiser par le Deep Jam

```
1 pour i : 0→N
2     pour j : 0→N
3         A_{i,j}
4     fin pour
5 fin pour
```

Listing 5.3 – Boucle à optimiser par le Deep Jam

```
1 pour i : 0→N
2     pour j : 0→N
3         si c_{1_i}
4         faire
5             A_{i,j}
6         fin si
7     fin pour
8 fin pour
```

Ensuite, il nous faut clarifier la manière de choisir la valeur du pas de la boucle servant pour la vectorisation (DJF), c'est-à-dire réfléchir et trouver une manière de décider quel facteur serait le mieux adapté à la vectorisation pour un programme donné. En effet, l'étude des microbenchmark nous avait amenés à penser que ce facteur était lié à la taille d'un registre vectoriel , alors que sur HydroMM cela ne semblait pas être le cas. Ce qui nous pousse à expérimenter sur d'autre code et sur des architectures offrant différentes tailles de registres vectoriels. Par exemple, nous pouvons faire varier le compilateur et sa version ou encore l'architecture cible.

Ce qui nous amène à étendre notre réflexion sur d'autres codes pour valider le passage à l'échelle que peut apporter notre variante de Deep Jam. Il apparait comme logique qu'il est plus intéressant que le benchmark offre du flot de contrôle irrégulier. Un exemple d'une application intéressante serait Lulesh qui offre beaucoup d'embranchements.

Appliquer l'algorithme à un code conséquent tel que Lulesh [1] nécessite d'abord de l'implémenter dans un compilateur, nous choisirons GCC. Ceci se fera en rajoutant une passe en gimple après la phase de création du CFG (graphe du flot de contrôle). Plus précisément, cette passe isole les blocs de base et les arrêtes le flot de contrôle (branchements, appels de fonctions, etc.) comme cela est visible sur le Listing 5.5 où des sections de type `<bb*>` ont été ajoutées pour délimiter les blocs de base. Les variables de type `D.2718` sont des variables locales créé par gimple. Il est donc plutôt aisé d'appliquer notre algorithme en plongeant dans le graphe. Il y a dans gimple un outil le « LNO » (Loop Nest Optimizer) qui permet de parcourir les boucles, de rentrer dans les nids et de réaliser des opérations sur des boucles telles que interchange ou encore hoisting. Nous pouvons l'utiliser pour réaliser les transformations de boucles. La passe et les transformations pourront être codées dans un plug-in.

1. `https://codesign.llnl.gov/lulesh.php`

Listing 5.4 – code gimple avant la passe CFG

```
main() {
    int D.2720;
    int x;
    x = 10 ;
    if (x!=0) goto <D.2718>;
    else goto <D.2719>;
    <D.2718>:
    {
        int y;
        y=5;
        D.2720 = x*y;
        x = D.2720+15
    }
    <D.2719>:
}
```

Listing 5.5 – code gimple après la passe CFG

```
main() {
    int y;
    int x;
    int D.2720;
<bb2>:
    x=10;
    if (x!=0) goto <bb 3>;
    else goto <bb 4>;
<bb3>:
    y=5;
    D.2720 = x*y;
    x=D.2720+15;
<bb4>:
    return;
}
```

5.2.2 Long terme

Même si l'intégration dans un compilateur de la gestion de notre algorithme pour le flot de contrôle n'est pas si éloignée, cela ne l'est pas pour le flot de données. En effet notre étude vis-à-vis du flot de données est relativement restreinte pour le moment. Nous n'avons pas dégagé d'heuristiques pour amener à la conception d'un autre algorithme. D'un point vu pratique cette implémentation dans GCC se fera en s'appuyant sur l'analyse de pointeur en détectant les indirections et en instrumentant celles-ci pour générer une trace à l'exécution. Le tout sera intégré dans un nouveau plug-in.

Nous pourrions aussi étudier l'impact sur les performances du Deep Jam sur un code déjà parallélisé en OpenMP (avec des `#pragma for` que des `#pragma task`), mais aussi potentiellement du parallélisme inter processus tel que MPI. Notre intuition nous laisse à penser qu'il n'y aura pas d'interférence. Nous avons réalisé quelques essais avec HydroMM avant de nous focaliser sur la vectorisation toute seule. Normalement, l'hybridation OpenMP Deep Jam est possible et valide au pire moyennant un Strip Mining. Mais il serait intéressant d'évaluer l'impact sur les performances.

Notre transformation étant non dépendante du langage du code, nous pourrions étendre aux autres langages supportés par GCC. Le plus utile serait le FORTRAN, car il est encore beaucoup utilisé dans les codes de calculs. En fait c'est parceque nos transformations se situeront dans le « middle-end » dans GCC avec la représentation intermédiaire gimple. Les benchmarks NAS Parallel Benchmarks (NPB) ont des kenels écrits en Fortran pour certain code de tests avec du flot de contrôle.

À plus longue échéance, nous pourrions vouloir ajouter à notre algorithme un plug-in GCC pour la détection de points chauds via une instrumentation du code, ou plus précisément des boucles de calculs. Nous pouvons à nouveau utiliser les fonctions du LNO pour rajouter une passe pour utilisant cela pour instrumenter les boucles. De plus ceci pourrait aussi servir à déterminer le chemin à privilégier.

Glossaire

AVX Advanced Vector eXtension. 11

BOSCC Branches-On-Superword-Condition-Codes. 28–30

DJF Facteur de Deep Jam. 48, 52, 54, 55, 57, 71, 73, 76, 77

GPGPU carte graphique dédiée au HPC. 9, 10, 12
GPU carte graphique. 9

HPC Calcul Haute Performance. 1, 2, 10, 11, 24, 79

KNC KNight Corner. 12, 19, 70
KNF KNight Ferry. 12
KNH KNight Hill. 12
KNL KNight Landing. 12

MMX MultiMedia eXtensions. 10

OpenCL Open Computing Language. 27, 33, 34, 37

SIMD Single Instruction Multiple Data. vi, 10, 13, 19, 21, 22, 31–33, 36, 39, 46, 56, 70, 75
SM Stream Multiprocessor. 9
SSE Streaming SIMD Extensions. 10

Bibliographie

[1] John R ALLEN et Ken KENNEDY. *PFC : A program to convert Fortran to parallel form*. Rice University. Department of Mathematical Sciences, 1982.
(Cf. p. 30, 31.)

[2] Randy ALLEN et Ken KENNEDY. « Automatic translation of FORTRAN programs to vector form ». In : *ACM Trans. Program. Lang. Syst.* 9.4 (oct. 1987), p. 491–542. ISSN : 0164-0925. DOI : 10.1145/29873.29875. URL : http://doi.acm.org/10.1145/29873.29875.
(Cf. p. 14, 30, 31.)

[3] David F. BACON, Susan L. GRAHAM et Oliver J. SHARP. « Compiler Transformations for High-performance Computing ». In : *ACM Comput. Surv.* 26.4 (déc. 1994), p. 345–420. ISSN : 0360-0300. DOI : 10.1145/197405.197406. URL : http://doi.acm.org/10.1145/197405.197406.
(Cf. p. 41.)

[4] Michaela BARTH et al. *Best Practices Guide Intel Xeon Phi v0.1*. Rapp. tech. Prace et Capacites, 31 mar. 2013.

[5] Cédric BASTOUL. « Code Generation in the Polyhedral Model Is Easier Than You Think ». In : *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA : IEEE Computer Society, 2004, p. 7–16. ISBN : 0-7695-2229-7. DOI : 10.1109/PACT.2004.11. URL : http://dx.doi.org/10.1109/PACT.2004.11.
(Cf. p. 33.)

[6] Cédric BASTOUL. *Efficient code generation for automatic parallelization and optimization (long version)*. Rapp. tech. 2003.

[7] Cédric BASTOUL et Albert COHEN. *Efficient Code Generation for Automatic Parallelization and Optimization*. URL : http://djdeath.atr.free.fr/presentation_version_3.ppt.

[8] Cédric BASTOUL et al. *Putting Polyhedral Loop Transformations to Work*. Anglais. Rapport de recherche RR-4902. INRIA, 2003. URL : http://hal.inria.fr/inria-00071681.
(Cf. p. 33.)

[9] Mohamed-Walid BENABDERRAHMANE et al. « The Polyhedral Model Is More Widely Applicable Than You Think ». In : *Compiler Construction*. Sous la dir. de Rajiv GUPTA. T. 6011. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, p. 283–303. ISBN : 978-3-642-11969-9. DOI : 10.1007/978-3-642-11970-5_16. URL : http://dx.doi.org/10.1007/978-3-642-11970-5_16.

[10] A. BERNA, M. JIMENEZ et J. M. LLABERIA. *Source code transformations for efficient SIMD code generation*.

[11] Aart J. C. BIK et al. « Automatic Intra-Register Vectorization for the Intel® Architecture ». In : *Int. J. Parallel Program.* 30.2 (avr. 2002), p. 65–98. ISSN : 0885-7458. DOI : 10.1023/A:1014230429447. URL : http://dx.doi.org/10.1023/A:1014230429447.
(Cf. p. 24, 26.)

[12] Aart J. C. BIK et al. « Automatic Intra-register Vectorization for the Intel Architecture ». In : *Int. J. Parallel Program.* 30.2 (avr. 2002), p. 65–98. ISSN : 0885-7458. DOI : 10.1023/A:1014230429447. URL : http://dx.doi.org/10.1023/A:1014230429447.

[13] Uday Kumar Reddy BONDHUGULA. « Effective automatic parallelization and locality optimization using the polyhedral model ». AAI3325799. Thèse de doct. Columbus, OH, USA, 2008. ISBN : 978-0-549-76796-1.

[14] Uday BONDHUGULA et al. « A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler ». In : *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria : ACM, 2010, p. 343–352. ISBN : 978-1-4503-0178-7. DOI : 10.1145/1854273.1854317. URL : http://doi.acm.org/10.1145/1854273.1854317.

[15] Patrick CARRIBAULT, Albert COHEN et William JALBY. « Deep Jam : Conversion of Coarse-Grain Parallelism to Instruction-Level and Vector Parallelism for Irregular Applications ». In : *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. PACT '05. Washington, DC, USA : IEEE Computer Society, 2005, p. 291–302. ISBN : 0-7695-2429-X. DOI : 10.1109/PACT.2005.16. URL : http://dx.doi.org/10.1109/PACT.2005.16.
(Cf. p. 38, 39.)

[16] Patrick CARRIBAULT et al. « Deep Jam : Conversion of Coarse-Grain Parallelism to Fine-Grain and Vector Parallelism ». In : *Journal of Instruction-Level Parallelism* 9 (2007), p. 1–26.

[17] Thomas J. Watson IBM Research CENTER, F.E. ALLEN et J. COCKE. *A Catalogue of Optimizing Transformations*. 1971. URL : https://books.google.fr/books?id=oeXaZwEACAAJ.

[18] Martyn CORDEN. *Requirements for vectorizing loops with #pragma SIMD*. 2012. URL : http://software.intel.com/en-us/articles/requirements-for-vectorizing-loops-with-pragma-simd (visité le 29/03/2013).
(Cf. p. 70.)

[19] CRAY INC. *Timeline Cray*. URL : http://www.cray.com/Assets/PDF/CrayTimeline.pdf.
(Cf. p. 14.)

[20] H. Carter EDWARDS et Daniel SUNDERLAND. « Kokkos Array Performance-portable Manycore Programming Model ». In : *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '12. New Orleans, Louisiana : ACM, 2012, p. 1–10. ISBN : 978-1-4503-1211-0. DOI : 10.1145/2141702.2141703. URL : http://doi.acm.org/10.1145/2141702.2141703.

[21] H. Carter EDWARDS et Christian R. TROTT. « Kokkos : Enabling Performance Portability Across Manycore Architectures ». In : *Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013)*. XSW '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 18–24. ISBN : 978-1-4799-3691-5. DOI : 10.1109/XSW.2013.7. URL : http://dx.doi.org/10.1109/XSW.2013.7.

[22] H. Carter EDWARDS, Christian R. TROTT et Daniel SUNDERLAND. « Kokkos : Enabling manycore performance portability through polymorphic memory access patterns ». In : *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, p. 3202–3216. ISSN : 0743-7315. DOI : http://dx.doi.org/10.1016/j.jpdc.2014.07.003. URL : http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[23] Alexandre E. EICHENBERGER, Peng WU et Kevin O'BRIEN. « Vectorization for SIMD Architectures with Alignment Constraints ». In : *SIGPLAN Not.* 39.6 (juin 2004), p. 82–93. ISSN : 0362-1340. DOI : 10.1145/996893.996853. URL : http://doi.acm.org/10.1145/996893.996853.
(Cf. p. 35, 36.)

[24] Pierre ESTÉRIE et al. « Boost.SIMD : Generic Programming for Portable SIMDization ». In : *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA : ACM, 2012, p. 431–432. ISBN : 978-1-4503-1182-3. DOI :

10.1145/2370816.2370881. URL : http://doi.acm.org/10.1145/2370816.2370881.
(Cf. p. 23.)

[25] T. GASC et al. « Modélisation de la performance et optimisation d'un algorithme hydrodynamique de type Lagrange-Projection sur processeurs multi-coeurs ». In : Presenté au Séminaire Aristote, 2015.
(Cf. p. 59.)

[26] Ronald W. GREEN. *Outer Loop Vectorization*. 2012. URL : http://software.intel.com/en-us/articles/outer-loop-vectorization.

[27] Christopher HAINE et al. « Exploring and Evaluating Array Layout Restructuration for SIMDization ». In : *The 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2014)*. Intel Corporation. Hillsboro, United States, sept. 2014. URL : https://hal.inria.fr/hal-01070467.
(Cf. p. 38.)

[28] Mark HAMPTON et Krste ASANOVIC. « Compiling for Vector-thread Architectures ». In : *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '08. Boston, MA, USA : ACM, 2008, p. 205–215. ISBN : 978-1-59593-978-4. DOI : 10.1145/1356058.1356085. URL : http://doi.acm.org/10.1145/1356058.1356085.

[29] Mark HARRIS. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/. Blog. 2013.

[30] INTEL INC. *Intel® Xeon Phi™ Coprocessor*. URL : http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[31] Gangwon JO et al. « OpenCL Framework for ARM Processors with NEON Support ». In : *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. Orlando, Florida, USA : ACM, 2014, p. 33–40. ISBN : 978-1-4503-2653-7. DOI : 10.1145/2568058.2568064. URL : http://doi.acm.org/10.1145/2568058.2568064.
(Cf. p. 37, 38.)

[32] Ralf KARRENBERG. « Automatic SIMD vectorization of SSA-based control flow graphs ». ger. Thèse de doct. Postfach 151141, 66041 Saarbraken : Universitat des Saarlandes, 2014. URL : http://scidok.sulb.uni-saarland.de/volltexte/2015/6096.

[33] Ralf KARRENBERG et Sebastian HACK. « Improving Performance of OpenCL on CPUs ». In : *Proceedings of the 21st International Conference on Compiler Construction*. CC'12. Tallinn, Estonia : Springer-Verlag, 2012, p. 1–20. ISBN : 978-3-642-28651-3. DOI : 10.1007/978-3-642-28652-0_1. URL : http://dx.doi.org/10.1007/978-3-642-28652-0_1.

[34] Ralf KARRENBERG et Sebastian HACK. « Whole-function Vectorization ». In : *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA : IEEE Computer Society, 2011, p. 141–150. ISBN : 978-1-61284-356-8. URL : http://dl.acm.org/citation.cfm?id=2190025.2190061.
(Cf. p. 34, 35.)

[35] Ken KENNEDY et Kathryn S. MCKINLEY. « Languages and Compilers for Parallel Computing : 6th International Workshop Portland, Oregon, USA, August 12–14, 1993 Proceedings ». In : sous la dir. d'Utpal BANERJEE et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. Chap. Maximizing loop parallelism and improving data locality via loop fusion and distribution, p. 301–320. ISBN : 978-3-540-48308-3. DOI : 10.1007/3-540-57659-2_18. URL : http://dx.doi.org/10.1007/3-540-57659-2_18.
(Cf. p. 35.)

[36] Changkyu KIM et al. *Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology*. Rapp. tech. Intel Labs.
(Cf. p. 35.)

[37] Seonggun KIM et Hwansoo HAN. « Efficient SIMD Code Generation for Irregular Kernels ». In : *SIGPLAN Not.* 47.8 (fév. 2012), p. 55–64. ISSN : 0362-1340. DOI : 10.1145/2370036.2145824. URL : http://doi.acm.org/10.1145/2370036.2145824.
(Cf. p. 35, 36.)

[38] L. KOESTERKE et al. « Early Experiences with the Intel Many Integrated Cores Accelerated Computing Technology ». In : (2011).

[39] C. KOZYRAKIS et al. « Hardware/compiler codevelopment for an embedded media processor ». In : *Proceedings of the IEEE* 89.11 (nov. 2001), p. 1694–1709. ISSN : 0018-9219. DOI : 10.1109/5.964446.

[40] Peter KRISTOF et al. « Performance Study of SIMD Programming Models on Intel Multicore Processors. » In : *IPDPS Workshops*. IEEE Computer Society, 2012, p. 2423–2432. ISBN : 978-1-4673-0974-5. URL : http://dblp.uni-trier.de/db/conf/ipps/ipdps2012w.html#KristofYLT12.
(Cf. p. 21.)

[41] Samuel LARSEN et Saman AMARASINGHE. « Exploiting Superword Level Parallelism with Multimedia Instruction Sets ». In : *SIGPLAN Not.* 35.5 (mai 2000), p. 145–156. ISSN : 0362-1340. DOI : 10.1145/358438.349320. URL : http://doi.acm.org/10.1145/358438.349320.
(Cf. p. 25.)

[42] Marco LATTUADA et Fabrizio FERRANDI. « Exploiting Outer Loops Vectorization in High Level Synthesis ». English. In : *Architecture of Computing Systems – ARCS 2015*. Sous la dir. de Luís Miguel Pinho PINHO et al. T. 9017. Lecture Notes in Computer Science. Springer International Publishing, 2015, p. 31–42. ISBN : 978-3-319-16085-6. DOI : 10.1007/978-3-319-16086-3_3. URL : http://dx.doi.org/10.1007/978-3-319-16086-3_3.

[43] Roland LEISSA, Immanuel HAFFNER et Sebastian HACK. « Sierra : A SIMD Extension for C++ ». In : *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. Orlando, Florida, USA : ACM, 2014, p. 17–24. ISBN : 978-1-4503-2653-7. DOI : 10.1145/2568058.2568062. URL : http://doi.acm.org/10.1145/2568058.2568062.
(Cf. p. 22, 23.)

[44] E. LINDHOLM et al. « NVIDIA Tesla : A Unified Graphics and Computing Architecture ». In : *Micro, IEEE* 28.2 (2008), p. 39–55. ISSN : 0272-1732. DOI : 10.1109/MM.2008.31.

[45] Scott Alan MAHLKE. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. Rapp. tech. 1996.

[46] Timothy Prickett MORGAN. *Intel slaps Xeon Phi brand on MIC coprocessors*. http://www.theregister.co.uk/2012/06/18/intel_mic_xeon_phi_cray/. Blog. 2012.
(Cf. p. 12.)

[47] Scott MUELLER. *Upgrading and Repairing PCs, 20th Edition*. August 16. 2011. ISBN : 978-0-7897-4710-5.
(Cf. p. 4, 5.)

[48] Dorit NAISHLOS et al. « Vectorizing for a SIMdD DSP Architecture ». In : *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA : ACM, 2003, p. 2–11. ISBN : 1-58113-676-5. DOI : 10.1145/951710.951714. URL : http://doi.acm.org/10.1145/951710.951714.

[49] C.J. NEWBURN et al. « Intel's Array Building Blocks : A retargetable, dynamic compiler and embedded language ». In : *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. Avr. 2011, p. 224–235. DOI : 10.1109/CGO.2011.5764690.

[50] Dorit NUZMAN et Ayal ZAKS. « Outer-loop vectorization : revisited for short SIMD architectures ». In : *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. PACT '08. Toronto, Ontario, Canada : ACM, 2008, p. 2–11. ISBN : 978-1-60558-282-5. DOI : 10.1145/1454115.1454119. URL : http://doi.acm.org/10.1145/1454115.1454119.

[51] Dorit NUZMAN et al. « Vapor SIMD : Auto-vectorize Once, Run Everywhere ». In : *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA : IEEE Computer Society, 2011, p. 151–160. ISBN : 978-1-61284-356-8. URL : http://dl.acm.org/citation.cfm?id=2190025.2190062.
(Cf. p. 25.)

[52] OPENACC REVIEW BOARD. *The OpenACC Application Programming Interface*. Version 1.0. Nov. 2011. URL : http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[53] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface*. Version 4.0 RC 2. Mar. 2013. URL : http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf.

[54] J.D. OWENS et al. « GPU Computing ». In : *Proceedings of the IEEE* 96.5 (2008), p. 879–899. ISSN : 0018-9219. DOI : 10.1109/JPROC.2008.917757.
(Cf. p. 9.)

[55] John D. OWENS et al. « A Survey of General-Purpose Computation on Graphics Hardware ». In : *Computer Graphics Forum* 26.1 (2007), p. 80–113. ISSN : 1467-8659. DOI : 10.1111/j.1467-8659.2007.01012.x. URL : http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x.
(Cf. p. 9.)

[56] Jérémy OZOG et David DUREAU. *Développement et évaluation d'un solveur CPU/GPGPU en hydrodynamique multi-matériaux*. Rapp. tech. CEA, DAM, DIF et INSA de Toulouse, juin 2011.
(Cf. p. 59.)

[57] Yongjun PARK et al. « SIMD Defragmenter : Efficient ILP Realization on Data-parallel Architectures ». In : *SIGARCH Comput. Archit. News* 40.1 (mar. 2012), p. 363–374. ISSN : 0163-5964. DOI : 10.1145/2189750.2151014. URL : http://doi.acm.org/10.1145/2189750.2151014.

[58] Yongjun PARK et al. « SIMD defragmenter : efficient ILP realization on data-parallel architectures ». In : *ASPLOS'12*. 2012, p. 363–374.

[59] Prakash PRABHU. *History of super computing, vector processing*. URL : http:/www.cs.princeton.edu/courses/archive/fall10/cos597C/docs/hardwaremod/History-of-SC-Vector-Processing.pptx.
(Cf. p. 13, 14, 19.)

[60] Fabien QUILLERE, Sanjay RAJOPADHYE et Doran WILDE. « Generation of Efficient Nested Loops from Polyhedra ». In : *International Journal of Parallel Programming* 28 (2000), p. 2000.

[61] James REINDERS. *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors*. Rapp. tech. Intel Labs, 2012. URL : http://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors.pdf.

[62] David ROBSON. *From CPU to GPU*. http://www.scientific-computing.com/hpcforscience/feature-gpu.html. Blog. 2009.

[63] N. ROTEM et Y. BEN ASHER. « Block Unification IF-conversion for High Performance Architectures ». In : *Computer Architecture Letters* 13.1 (jan. 2014), p. 17–20. ISSN : 1556-6056. DOI : 10.1109/L-CA.2012.28.
(Cf. p. 27.)

[64] Sean RUL et al. « An experimental study on performance portability of OpenCL kernels ». eng. In : *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*. Knoxville, TN, USA, 2010, p. 3. URL : https://biblio.ugent.be/publication/1016024/file/1016062.pdf.

[65] Thomas SCHAUB et al. « The Impact of the SIMD Width on Control-Flow and Memory Divergence ». In : *ACM Trans. Archit. Code Optim.* 11.4 (jan. 2015), 54 :1–54 :25. ISSN : 1544-3566. DOI : 10.1145/2687355. URL : http://doi.acm.org/10.1145/2687355.

[66] Ayal Zaks SHAHAR TIMNAT Ohad Shacham. « Predicate Vectors If You Must ». In : 2014. URL : https://sites.google.com/site/wpmvp2014/paper_8.pdf.
(Cf. p. 27.)

[67] Jaewook SHIN. « Introducing Control Flow into Vectorized Code ». In : *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA : IEEE Computer Society, 2007, p. 280–291. ISBN : 0-7695-2944-5. DOI : 10.1109/PACT.2007.41. URL : http://dx.doi.org/10.1109/PACT.2007.41.
(Cf. p. 28–30.)

[68] Jaewook SHIN, J. CHAME et Mary W. HALL. « Compiler-controlled caching in superword register files for multimedia extension architectures ». In : *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. 2002, p. 45–55. DOI : 10.1109/PACT.2002.1106003.

[69] Jaewook SHIN, Mary W. HALL et Jacqueline CHAME. « Evaluating Compiler Technology for Control-flow Optimizations for Multimedia Extension Architectures ». In : *Microprocess. Microsyst.* 33.4 (juin 2009), p. 235–243. ISSN : 0141-9331. DOI : 10.1016/j.micpro.2009.02.002. URL : http://dx.doi.org/10.1016/j.micpro.2009.02.002.
(Cf. p. 29.)

[70] Jaewook SHIN, Mary HALL et Jacqueline CHAME. « Superword-Level Parallelism in the Presence of Control Flow ». In : *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA : IEEE Computer Society, 2005, p. 165–175. ISBN : 0-7695-2298-X. DOI : 10.1109/CGO.2005.33. URL : http://dx.doi.org/10.1109/CGO.2005.33.
(Cf. p. 27, 28.)

[71] Jon S. SQUIRE et Sandra M. PALAIS. « Programming and design considerations of a highly parallel computer ». In : *Proceedings of the May 21-23, 1963, spring joint computer conference*. AFIPS '63 (Spring). Detroit, Michigan : ACM, 1963, p. 395–400. DOI : 10.1145/1461551.1461597. URL : http://doi.acm.org/10.1145/1461551.1461597.
(Cf. p. 13.)

[72] Xinmin TIAN et al. « Practical SIMD Vectorization Techniques for Intel®Xeon Phi&#153 ; Coprocessors ». In : *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 1149–1158. ISBN : 978-0-7695-4979-8. DOI : 10.1109/IPDPSW.2013.245. URL : http://dx.doi.org/10.1109/IPDPSW.2013.245.
(Cf. p. 36, 37.)

[73] Konrad TRIFUNOVIĆ et al. « Polyhedral-Model Guided Loop-Nest Auto-Vectorization ». In : *The 18th International Conference on Parallel Architectures and Compilation Techniques*. Raleigh, United States, sept. 2009. URL : https://hal.inria.fr/hal-00645325.

[74] Peng WU, A.E. EICHENBERGER et A. WANG. « Efficient SIMD code generation for runtime alignment and length conversion ». In : *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*. Mar. 2005, p. 153–164. DOI : 10.1109/CGO.2005.18.

[75] Peng WU et al. « An Integrated Simdization Framework Using Virtual Vectors ». In : *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts : ACM, 2005, p. 169–178. ISBN : 1-59593-167-8. DOI : 10.1145/1088149.1088172. URL : http://doi.acm.org/10.1145/1088149.1088172.
(Cf. p. 31, 32.)

[76] Jinlong XU, Huihui SUN et Rongcai ZHAO. « SIMD vectorization of nested loop based on strip mining ». In : *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*. Juin 2015, p. 1–7. DOI : 10.

1109/SNPD.2015.7176176.
(Cf. p. 32, 33.)