Modèle de programmation bas niveau pour architecture de calcul proche mémoire

Kévin Mambu

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'intégration de systèmes et de technologies

Modèle de programmation bas niveau pour architecture de calcul proche mémoire

Low-level programming model for In-Memory Computing architecture

Présentée par :

Kévin MAMBU

Direction de thèse :

Henri-Pierre CHARLES Directeur de recherche, CEA Centre de Grenoble	Directeur de thèse
Maha KOOLI Ingénieur-Chercheur, CEA Grenoble	Co-encadrant de thèse

Rapporteurs :

LIONEL LACASSAGNE
Professeur des Universités, SORBONNE UNIVERSITE
ALBERTO BOSIO
Professeur, ECOLE CENTRALE LYON

Thèse soutenue publiquement le **10 mars 2023**, devant le jury composé de :

HENRI-PIERRE CHARLES Directeur de recherche, CEA CENTRE DE GRENOBLE	Directeur de thèse
CAROLINE COLLANGE Chargé de recherche HDR, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE	Examinatrice
LAURE GONNORD Professeur des Universités, GRENOBLE INP	Examinatrice
FREDERIC PETROT Professeur des Universités, GRENOBLE INP	Examinateur
LIONEL LACASSAGNE Professeur des Universités, SORBONNE UNIVERSITE	Rapporteur
ALBERTO BOSIO Professeur, ECOLE CENTRALE LYON	Rapporteur

Invités :

MAHA KOOLI
Ingénieur de recherche, CEA CENTRE DE GRENOBLE

À mes parents et mes amis proches,

Remerciements

Lorsque j'ai démarré cette thèse, je me suis trouvé de prime abord intimidé par l'ampleur de la portée du sujet de recherche. Mais j'ai eu la chance et le privilège de faire partie d'un environnement de travail qui m'a galvanisé et motivé à pousser jusqu'au bout ce qui est essentiellement mon premier projet de recherche (et, j'y songe bien, pas le dernier). C'est ainsi que je me dois de remercier tous les membres de la *Memory Design Team*, pour leur accueil ainsi que pour leur convivialité. Je ne peux que me remémorer avec un sourire aux lèvres les discussions intéressantes, les conseils pertinents et le soutien moral que vous m'avez apporté. De tous nos échanges je ressors – j'espère – grandi d'une plus profonde compréhension du domaine de l'architecture électronique et de la programmation informatique, qui me servira de bagage fondamental pour le reste de ma carrière.

Un grand merci également à mon directeur de thèse, Henri-Pierre CHARLES, pour avoir défendu l'ouverture de ce sujet de thèse, et pour m'avoir introduit à ce domaine fantastique qu'est la compilation dynamique. Si je dois bien admettre, que ce concept m'a rendu dubitatif par instants, force est de constater que ce domaine présente beaucoup de potentiel et d'intérêt, à l'heure où j'écris ces lignes, et je me dois de te porter mes respects pour ta direction visionnaire et l'ambition à laquelle tu m'as laissé l'occasion de contribuer. Merci également mon encadrante de thèse, Maha KOOLI, pour tes conseils et pour le support que tu m'as apporté même lors de périodes surchargées. Tu m'as été une source d'inspiration quand à mon savoir-être professionnel.

Je tiens également à remercier mes parents (Maman, Gaël, Papa et Sedami) qui m'ont toujours apporté leur soutien et leur amour. Un grand merci également à tous mes amis proches pour toutes ces discussions et tout ce soutien, dans des périodes où je doutais de réussir à aller jusqu'au bout. C'est grâce à vous toutes et tous que je suis arrivé jusqu'ici, et je ne vous oublierai certainement pas pour cela.

Et enfin, un remerciement tout particulier au Dr. Pirouz BAZARGAN-SABET, qui m'a fait découvrir les architectures des ordinateurs, et qui m'a soutenu à une période où je n'étais pas sûr de poursuivre mes études. Vos encouragements m'ont donné à l'époque la force de poursuivre de la Licence au Master, et du Master au Doctorat. Je vous en suis éternellement reconnaissant.

Résumé

Depuis les années 60 le modèle architectural utilisé par les processeurs est le modèle 'von Neumann' dans lequel un processeur va chercher instructions et données à traiter dans la même mémoire. L'augmentation de la densité de transistor sur une puce a permis d'augmenter sa fréquence de fonctionnement mais a produit un 'goulot d'étranglement' vers la mémoire qui ne peut pas fournir instructions et données à la même fréquence : le mur de la mémoire.

Beaucoup de solutions architecturales ont été proposées pour résoudre ce goulot d'étranglement. Une des solutions que nous étudions est une architecture dans laquelle les calculs sont réalisés dans la mémoire, sans déplacer les données vers le processeur. L'évaluation de cette solution a montré des gains potentiels impressionnants en vitesse et en énergie.

Pour exploiter ce potentiel il faut changer de modèle de programmation car les instructions ne seront plus lues en mémoire mais générées par un processeur qui pilotera un ou plusieurs plans mémoire.

Les contributions de la thèse sont la spécification d'un mécanisme de transfert pour les motifs d'accès mémoire complexes à destination des architectures de calcul proche-mémoire. La thèse présente également un modèle de programmation haut-niveau permettant la programmation d'une architecture de calcul proche-mémoire, ainsi que du mécanisme de transfert susmentionné. Ce modèle de programmation peut être paramétré pour compiler des applications spécialisées grâce à la compilation statique, ou la génération dynamique de code pour effectuer des optimisations lors du run-time.

L'évaluation de ces contributions par le biais d'un modèle de simulation montre des résultats qui témoignent de l'intérêt de la spécialisation dynamique de code pour les architectures de calcul proche-mémoire, et par extension des nœuds de calcul hétérogènes.

Abstract

Since the 60's the architectural model used by processors is the 'von Neumann' model in which a processor will look for instructions and data to be processed in the same memory. The increase of the transistor density on a chip has allowed to increase its operating frequency but has produced a 'bottleneck' towards the memory which cannot provide instructions and data at the same frequency : the memory wall.

Many architectural solutions have been proposed to solve this bottleneck. One of the solutions we are studying is an architecture in which computations are performed in memory, without moving data to the processor. Evaluation of this solution has shown impressive potential gains in speed and energy.

To exploit this potential, a change in programming model is required, as instructions will no longer be read from memory but generated by a processor that will drive one or more memory planes.

The contributions of the thesis are the specification of a transfer mechanism for complex memory access patterns for near-memory computing architectures. The thesis also presents a high-level programming model allowing the programming of a near-memory computing architecture, as well as the above-mentioned transfer mechanism. This programming model can be parameterized to compile specialized applications through static compilation, or dynamic code generation to perform run-time optimizations.

The evaluation of these contributions through a simulation model shows results that demonstrate the interest of dynamic code specialization for near-memory computing architectures, and by extension heterogeneous computing nodes.

Table des matières

Introduction générale

Contexte et objectifs

Une grande majorité des architectures de calcul sont basées sur le modèle architectural de von Neumann. Dans ce modèle, un *Central Processing Unit* (CPU) récupère des instructions et transfère des données depuis une mémoire principale.

Cependant, le modèle von Neumann souffre par ce choix d'organisation de limitations de performance inhérentes à la mémoire principale. Ces limitations font de cette dernière un goulot d'étranglement aussi bien en termes de temps d'exécution qu'en efficacité énergétique.

Aujourd'hui, de nombreux domaines applicatifs tels que le Traitement d'image ou le *Deep Learning* requièrent d'importantes quantités de données. Cependant, l'évolution technologique des semi-conducteurs rend le *bottleneck* de von Neumann toujours plus critique dans la performance des applications. Cette limitation rend nécessaire la mise au point de nouveaux paradigmes architecturaux.

L'*In-Memory Computing* (IMC) tente de résoudre ce problème de bottleneck par l'intégration d'unités de calcul au sein d'unités mémoires. Ainsi, les transferts de données entre le *Host Processing Unit* (HPU) et les mémoires IMC sont réduites, car ne nécessitant plus d'échanges de données de calcul mais l'envoi de requêtes d'opérations vectorielles vers l'IMC. Il existe de multiples solutions IMC, qui diffèrent selon des critères tels que les technologies mémoires sur lesquelles elles sont basées, ainsi que leur capacité de calcul. Toutes ces solutions présentent des résultats pertinents et qui montrent du potentiel en l'IMC pour être une alternative viable dans l'implémentation d'architectures de calcul non-von Neumann.

Néanmoins, l'intégration de l'IMC dans d'actuelles architectures de calcul introduit de nouveaux paradigmes de programmation et de nouvelles problématiques spécifiques à ces derniers. Bien qu'il soit possible d'adapter un modèle de programmation existant de la littérature pour architectures hétérogènes tels que les *Graphics Processing Unit* (GPU), une adéquation entre le matériel pour lequel le modèle était initialement prévu et l'IMC est requise. Cette adéquation

n'est pourtant pas toujours présente, car la conception architecturale de l'IMC peut varier d'une proposition à l'autre. De plus, des problématiques exclusives à l'IMC, tels que l'organisation et l'alignement des données, ainsi que la consommation de bande passante au niveau d'un système mémoire complet, ne sont pas nécessairement traitées par des modèles de programmation antérieurs. Il est donc pertinent de mettre en place un modèle de programmation dédié afin de bénéficier efficacement des fonctionnalités de l'IMC.

Les objectifs de cette thèse sont :

— Proposer un modèle de programmation pour une architecture non-von Neumann – en particulier une architecture de calcul proche-mémoire IMC.

— Développer une méthodologie d'évaluation pour valider l'intégration de l'architecture au sein d'un système complet, composé d'une ou plusieurs unités de calcul et une ou plusieurs unités mémoire.

Cette thèse s'intègre dans des travaux autour de :

— La conception de la *Computational SRAM* (C-SRAM), une architecture IMC basée sur la technologie mémoire SRAM qui permet d'effectuer du calcul vectoriel au sein de la mémoire pour réduire la consommation de bande passante entre cette dernière et le HPU.

— Le développement de Hybrogen, un nouvel environnement de compilation dédié à la génération dynamique de code.

Les travaux de la présente thèse ont été développés au sein du *Laboratoire de Fonctions Innovantes pour circuits Mixtes* (LFIM) du CEA Grenoble, dont le but est d'élaborer et vérifier des solutions logicielles et matérielles dédiées au calcul hétérogène. Ce laboratoire fait partie de l'institut LIST de recherche technologique, et de la Direction de Recherche Technologique (DRT) du CEA Grenoble.

Contributions

Les contributions de la thèse sont les suivantes :

1. La spécification du *Data-locality Management Unit* (DMU), un contrôleur mémoire spécialisé pour transférer de façon efficace les données pour les codes de stencil, une classe d'applications couramment employés dans les applications numériques.
2. La spécification et l'implémentation de *IMCCC*, un modèle de programmation dédié à la mémoire C-SRAM et au contrôleur DMU. Le modèle IMCCC est basé sur la syntaxe du langage C et permet de décrire des programmes implicitement vectoriels pour pouvoir exprimer des motifs d'accès mémoire de stencils, et les associer à la programmation du contrôleur DMU.
3. La proposition d'une méthodologie d'inter-opération entre IMCCC et Hybrogen, pour permettre la programmation d'applications C-SRAM employant la compilation dynamique.
4. La spécification d'une méthodologie de modélisation et de simulation de systèmes de calcul intégrant la mémoire C-SRAM et le contrôleur DMU, pour validation architecturale et logicielle, basée sur l'émulateur QEMU.

Présentation du contenu de la thèse

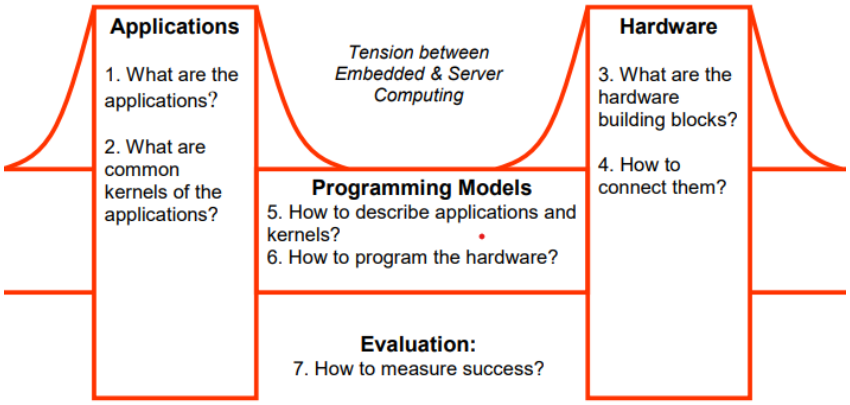


FIGURE 0.1 – Les sept questions scientifiques retenues par l'Université de Berkeley, quand au futur de la programmation parallèle. Image tirée de ([ABC+06]).

La communauté scientifique de l'université de Berkeley a énuméré un ensemble de questions scientifiques, jugées critiques pour la haute performance des architectures émergentes :

$Q1$: Quelles sont les applications nécessaires pour la résolution de problèmes numériques et arithmétiques ?

$Q2$: Quels sont les noyaux de calcul – les opérations fondamentales – qui composent ces applications ?

$Q3$: Quels sont les composants matériels nécessaires pour l'exécution de ces noyaux de calcul ?

$Q4$: Comment interfacer ces composants ?

$Q5$: Comment décrire à haut-niveau les applications et les noyaux de calcul ?

$Q6$: Comment programmer le matériel pour exécuter les applications décrites ?

$Q7$: Comment déterminer que l'exécution du logiciel sur le matériel est satisfaisante ?

La Figure 0.1 présente ces questions scientifiques sous la forme des dynamiques d'interaction entre le logiciel et le matériel. Entre la conception des architectures de calcul et l'élaboration des applications, se trouve les modèles de programmation. Ces modèles sont essentiels pour interfacer les programmeurs avec le matériel, et leur conception présente également un ensemble de problématiques au-delà de celles soulevées par les questions sus-mentionnées.

Nous basons notre méthodologie de recherche sur la base de ces questions scientifiques afin de proposer de nouvelles solutions à l'art existant. La présente thèse se décompose comme suit :

— Le Chapitre 1 présente de façon non exhaustive l'État de l'art concernant l'évolution des architectures de calcul, selon une approche historique. L'objectif de ce chapitre est de mettre en contexte les architectures de calcul proche-mémoire vis-à-vis de l'État de l'art. Nous apporterons ainsi des éléments antérieurs de réponse aux questions $Q3$ et $Q4$.

— Le Chapitre 3 présente les défis logiciels quant à la programmation de systèmes de calcul contenant des nœuds de calcul hétérogènes. Nous apporterons ainsi des éléments antérieurs de réponse aux questions $Q1$ et $Q2$.

— Le Chapitre 4 introduit notre première contribution : le *Data-locality Management Unit*, un contrôleur mémoire spécialisé pour transférer de façon efficace des données de stencil vers la mémoire C-SRAM. Cette contribution apporte un nouvel élément de réponse aux questions $Q3$, $Q4$ et $Q6$.

— Le Chapitre 5 introduit notre seconde contribution : *IMCCC*, un modèle de programmation pour la mémoire C-SRAM et le contrôleur DMU. Ce modèle de programmation permet de décrire des applications intégrant des codes de stencil de façon efficace. Cette contribution apporte un nouvel élément de réponse aux questions $Q5$ et $Q6$.

— Enfin, le Chapitre 6 présente notre méthodologie d'évaluation d'applications utilisant la mémoire C-SRAM et le contrôleur DMU. La méthodologie se base sur la modélisation et la simulation de systèmes de calcul qui intègrent ces deux composants. Cette contribution apporte un nouvel élément de réponse à la question $Q7$.

Publications

Julie Dumas, Henri-Pierre Charles, Kévin Mambu, Maha Kooli. *Dynamic Compilation for Transprecision Applications on Heterogeneous Platform.* MDPI Journal of Low Power Electronics and Applications (JLPEA) 11, no. 3. Accepté et publié en Juin 2021. Présentation de l'environnement Hybrogen pour la compilation dynamique de code sur architectures de calcul hétérogène.

Maha Kooli, Antoine Heraud, Henri-Pierre Charles, Bastien Giraud, Roman Gauchi, Mona Ezzadeen, Kévin Mambu, Valentin Egloff, and Jean-Philippe Noël. *Towards a Truly Integrated Vector Processing Unit for Memory-bound Applications Based on a Cost-competitive Computational SRAM Design Solution.* ACM Journal of Emerging Technologies and Computing Systems (JETC) 18, 2, Article 40. Accepté en Septembre 2021, publié en Avril 2022. Présentation de la spécification et la conception de la mémoire SRAM Computationelle (C-SRAM).

Kevin Mambu, Julie Dumas, Henri-Pierre Charles, Maha Kooli. *Instruction Set Design Methodology for In-Memory Computing through QEMU-based System Emulator.* 32nd International Workshop on Rapid System Prototyping (RSP). Accepté en Octobre 2021, publié en Juin 2022. Présentation d'un flot conjoint de compilation, de modélisation et de simulation de jeux d'instruction pour les architectures de calcul proche-mémoire, basée sur QEMU. L'article contient une expérimentation pour la validation des modèles.

Kévin Mambu, Henri-Pierre Charles, Maha Kooli, and Julie Dumas. *Towards Integration of a Dedicated Memory Controller and Its Instruction Set to Improve Performance of Systems Containing Computational SRAM.* MDPI Journal of Low

Power Electronics and Applications 12, no. 1 : 18. Accepté en Février 2022, publié en Mars 2022. Spécification et évaluation d'un contrôleur mémoire spécialisé pour les architectures de calcul proche-mémoire. L'article présente l'évaluation de la solution proposée grâce à sa modélisation au sein de QEMU.

Kévin Mambu, Henri-Pierre Charles, Maha Kooli. *Dedicated Instruction Set for Pattern-based Data Transfers : an Experimental Validation on Systems Containing In-Memory Computing Units*. IEEE Transactions on Computer Design and Automation (TCAD). Accepté avec révisions mineures. Présentation de l'intégration d'un contrôleur mémoire spécialisé au sein du jeu d'instruction des architectures de calcul proche-mémoire, et spécification d'un encodage spécialisé pour les transferts de voisinages de stencil. L'article présente l'évaluation d'une telle intégration sur trois architectures de calcul basées sur des CPUs RISC-V.

Brevets

Kévin Mambu, Henri-Pierre Charles, Maha Kooli. *Système de transfert direct de données #1*. Déposé à l'Institut National de la Propriété Intellectuelle (INPI). Établissement du jeu d'instructions du DMU et de son interface avec le jeu d'instructions de la C-SRAM.

Kévin Mambu, Henri-Pierre Charles, Maha Kooli. *Système de transfert direct de données #2*. Déposé à l'échelle internationale. Établissement du format d'encodage binaire des voisinages de stencil (intitulé *Pattern Stream Encoding* lors de communications scientifiques).

Chapitre

1 Limitations du modèle architectural de von Neumann

- *Quels sont les composants matériels existants ?*
- *Comment sont interfacés ces composants ?*

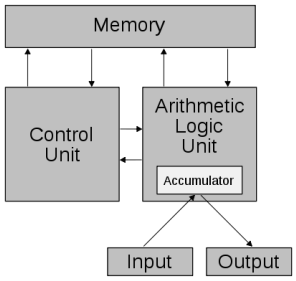
Introduction

Dans ce chapitre, nous présentons les limitations du modèle architectural de von Neumann, les solutions de l'art antérieur pour mitiger ces limitations et les modèles architecturaux alternatifs explorés pour outrepasser le *goulot d'étranglement de von Neumann*.

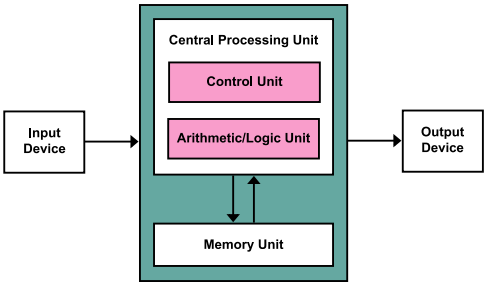
Ce chapitre est décomposé comme suit : section 1.1 présente et caractérise le *Bottleneck de von Neumann*, ou le goulot d'étranglement à l'origine de la limitation de la performance des architectures basées sur ce modèle. Section 1.2 présente quelques solutions incorporées initialement dans les architectures basées sur le modèle de von Neumann pour mitiger le phénomène de goulot d'étranglement. Section 1.3 introduit des paradigmes architecturaux alternatifs au modèle architectural de von Neumann, par l'intégration de topologies mémoires distribuées et de calcul hétérogène. Enfin, section 1.3.3 conclut ce chapitre.

1.1 Caractérisation du *Bottleneck de von Neumann*

Le modèle architectural de von Neumann est un modèle d'organisation architectural des machines de calcul, attribué entre autres à John von Neumann et l'équipe à l'origine du développement du *Electronic Discrete Variable Automatic Computer* (EDVAC), mis en service en 1951[VN93a]. Les architectures de von Neumann se caractérisent par leur organisation architecturale, c'est-à-dire les composants en leur sein et les interfaces de connexion de ces dernières.



(a) Modèle de von Neumann original, selon [VN93b].



(b) Version moderne et simplifiée (source : Wikimedia Commons).

FIGURE 1.1 – Representations du modèle architectural de von Neumann.

Les Figures 1.1a et 1.1b représentent toutes deux une architecture de von Neumann, chacune avec une organisation architecturale similaire. Nous pouvons observer deux interfaces d'Entrées/-Sorties (E/S) pour permettre au programmeur de communiquer. Ces E/S sont connectées à une *Unité Arithmétique et Logique*, ou encore *Arithmetic-Logic Unit* (ALU). L'ALU possède en son sein un ou plusieurs registres, des ressources de stockage pour les opérandes des calculs temporaires du programme à exécuter. Une *Unité de Contrôle*, ou encore *Control Unit* (CU), commande l'ALU pour effectuer les calculs par le biais d'instructions, qui composent le programme. Ces instructions, ainsi que les ressources matérielles de l'architecture qui sont accessibles aux programmeurs, composent une spécification d'interface logicielle/matérielle couramment nommée *Architecture de Jeu d'Instructions*, ou *Instruction Set Architecture* (ISA). La CU et l'ALU composent tous deux le composant que nous appelons aujourd'hui le *Central Processing Unit* (CPU). Enfin, une *Mémoire* va stocker les données temporaires qui ne peuvent pas être stockées dans le CPU, ainsi que les instructions du programme à exécuter.

Lors de la création de l'EDVAC, d'autres machines de calcul employaient des organisations architecturales différentes, telles que le IBM *Automated Sequence-Controlled Calculator* (ASCC) *Mark I*. Le ASCC Mark I sera à l'origine du modèle architectural dit *de Harvard*.

La Figure Fig. 1.2 fait la comparaison entre l'organisation architecturale d'une machine de von Neumann et d'une machine de Harvard. Nous pouvons observer qu'une machine de Harvard emploie deux mémoires physiquement distinctes : une mémoire d'instructions et une mémoire de données, tandis qu'une machine de von Neumann utilise une seule mémoire physique. Une machine de Harvard a comme avantage, par rapport à une machine de von Neumann, de pouvoir récupérer une donnée et une instruction en même temps, alors qu'une machine de von Neumann devra effectuer de façon séquentielle ces accès mémoires.

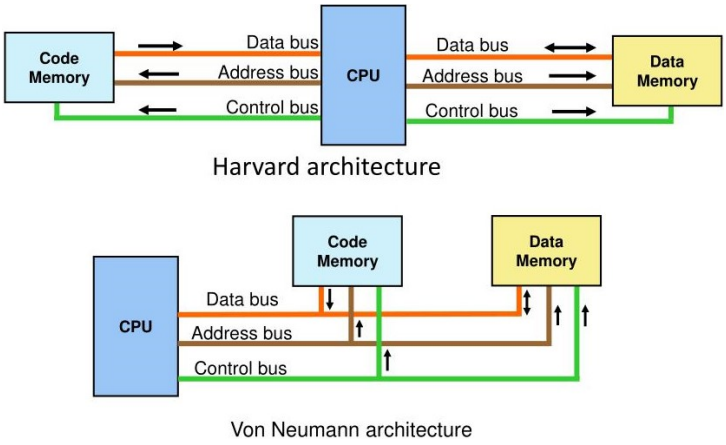
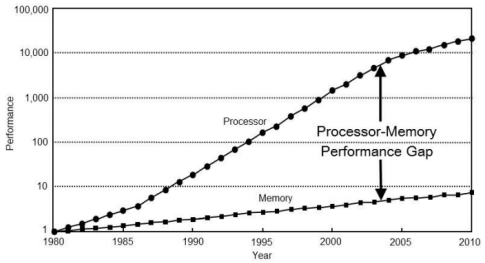


FIGURE 1.2 – Comparaison entre les modèles architecturaux historiques de Harvard et de von Neumann. La différence principale est la distribution des unités de mémoires instructions et données pour leur permettre des accès concurrents au même cycle d'horloge. Source : Wikimedia Commons.

Les architectures de von Neumann modernes ne disposent cependant pas de ces limitations dans la majorité des cas. En effet, elles peuvent émuler une architecture de Harvard via l'utilisation d'une mémoire logiquement séparée en deux régions : une d'instructions et une de données, qui dispose de deux ports de lecture/écriture pour accéder à ces deux régions de façon concurrente dans le même temps d'horloge. Pour ces raisons, le modèle von Neumann est resté pendant longtemps le modèle d'organisation architectural standard pour l'implémentation d'architectures de calcul généraliste.



(a) Évolution de l'écart de performance entre les CPUs et les mémoires principales, des années 1980 aux années 2000[HP11].

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

(b) Distribution de la consommation énergétique au sein d'une architecture CPU moderne, cités de [Hor14].

FIGURE 1.3 – Données de la littérature qui caractérisent le bottleneck de von Neumann en termes de latence d'opérations (*Memory Wall*) et d'efficacité énergétique (*Energy Wall*).

Cependant, ce modèle est soumis à des limitations intrinsèques liées aux performances de fonctionnement de la mémoire, qui donnent lieu à deux phénomènes :

— Le *Memory Wall*, ou la limitation de latence du CPU par rapport à celle de la mémoire.

Ce Memory Wall est devenu au fil des années de plus en plus conséquent, à a suite de la divergence entre l'évolution technologique des CPU et celle de la mémoire.

La Figure 1.3a, tirée de [HP11] illustre ce phénomène en montrant l'évolution de la performance relative des mémoires et des processeurs, des années 1980 aux années 2000. Nous pouvons observer que l'écart de performance entre ces deux types de composants devient de plus en plus lagre au cours de cette période.

— L'*Energy Wall*, ou l'écart entre le coût énergétique du transfert d'une donnée en opposition au coût de son calcul. Cet écart énergétique s'explique par les contraintes technologiques intrinsèques des mémoires, mais également par le surcoût du transfert à travers le bus de données.

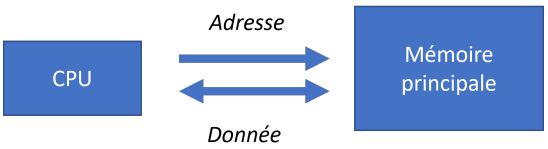
La Figure 1.3b, tirée de [Hor14] présente la décomposition de la consommation énergétique d'un processeur. Nous pouvons observer que l'écart entre la consommation énergétique d'une addition 32-bit et un accès à une mémoire DRAM peut s'élever à un facteur entre ×13000 à ×26000.

En conséquence de ces deux phénomènes, la performance des architectures de von Neumann est bornée par la performance de la mémoire. Au fur et à mesure que les applications sont devenues de plus en plus gourmandes en données, ce *goulot d'étranglement* de la performance des architectures de von Neumann est devenu un problème toujours plus critique à résoudre. C'est ainsi que de nombreuses solutions ont été proposées pour maintenir des organisations architecturales qui mitigent ces limitations, tout en présentant sur le plan logique – c'est-à-dire du point de vue des programmeurs et des logiciels – des architectures de von Neumann.

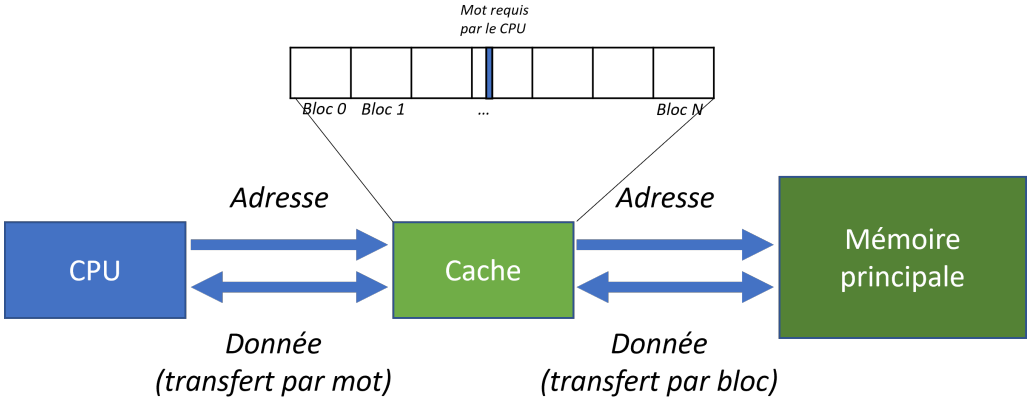
1.2 Mitigation du goulot d'étranglement de von Neumann

Afin d'améliorer la performance des applications, des systèmes de stockage plus complexes sont mis en place. L'intégration de **bancs de registres** permet au logiciel de maintenir davantage de données en mémoire que les seuls registres à accumulation des machines de calcul développées dans les années 1960. Les bancs de registres ont une très faible latence mais ne peuvent pas être utilisés en grande quantité pour au moins deux raisons. Tout d'abord, ces registres sont onéreux à intégrer dans une architecture de calcul car elles emploient avec des technologies de stockage onéreuses. De plus, l'adressage des bancs de registres a un impact sur la taille des instructions d'une ISA, et il faut donc trouver un compromis entre un nombre de registres adressables et la taille des instructions.

C'est pour cela qu'une nouvelle solution sera développée au cours du temps : les **mémoires cache**. Ces dernières utilisent une stratégie différente que les bancs de registre : au lieu d'avoir un impact direct sur une ISA, elles vont dissimuler autant que possible les latences d'accès du CPU à la mémoire principale.



(a) Architecture de von Neumann sans mémoire cache.



(b) Architecture de von Neumann avec mémoire cache.

FIGURE 1.4 – Les mémoires cache dissimulent la latence des accès CPU aux données grâce au chargement de données depuis la mémoire principale par blocs, et une plus faible latence d'accès.

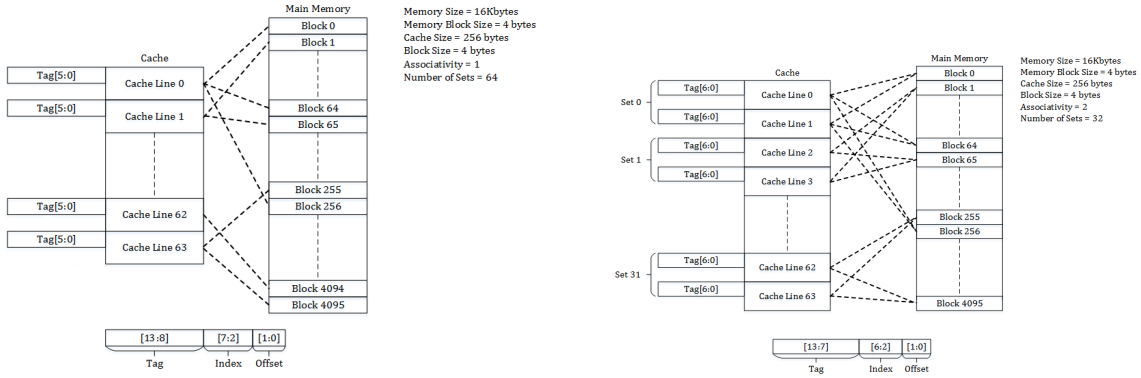
La Figure 1.4 présente une architecture de von Neumann avant, et après l'intégration d'une mémoire cache. La mémoire cache s'interface entre le CPU et la mémoire principale avec les mêmes ports d'entrée et de sortie pour ne pas requérir de modification matérielle ou logicielle de la part du CPU. La mémoire cache et la mémoire principale compose tous deux la *hiérarchie mémoire* du CPU, au sein de l'architecture de von Neumann. Au sein de la mémoire cache, les données sont organisées par lignes de cache.

Lorsqu'un mot de données est requis auprès d'une mémoire cache et que cette dernière est présente, on parle de *cache hit*. Dans le cas contraire, on parle de *cache miss*. Le contrôleur du cache va alors émettre une requête à la mémoire de niveau inférieur pour récupérer le *bloc de données* qui contient la donnée requise. Ce bloc de données est de la taille d'une *ligne de cache*, ce qui signifie que pour un cache avec des lignes de 64 octets, un cache miss à la lecture d'une donnée entraînera nécessairement la lecture au niveau inférieur du bloc de 64 octets contenant ladite donnée.

L'utilisation de transferts par blocs améliore la performance des accès mémoires émis par le CPU selon les principes suivants. Le premier est la *localité temporelle* des données, ou le fait que des données récemment transférées ont une forte probabilité d'être réutilisées dans un futur proche. Le deuxième est la *localité spatiale* des données, ou le fait que des éléments logiquement proches en mémoire auront tendance à être transférés de façon consécutive à des instants proches. Par ces deux principes, le chargement de données par blocs des contrôleurs de cache se révèle souvent très efficace pour des accès à des structures de données très régulières

tels que des séquences d'instructions contiguës et des tableaux de données.

Si les mémoires caches possèdent une plus grande capacité que les bancs de registres, elles restent cependant bien plus petites que la mémoire principale, bien qu'elles doivent être en mesure de pouvoir stocker l'ensemble de la plage de données stockées dans cette dernière. Pour ces raisons, des mécanismes de placement de données doivent être mis en place afin de savoir quelle région de la mémoire peut être stockée dans quelle ligne de cache. Lors de la réception d'une requête d'un mot de données le contrôleur de la mémoire cache va décomposer l'adresse reçue en entrée en plusieurs champs : l'*offset* qui correspond au numéro de l'octet requis au sein du bloc, l'*index* qui correspond au numéro de ligne de cache associé audit bloc, et le *tag* qui est l'entête de l'index. Ce tag sert d'identificateur du bloc couramment chargé dans la mémoire cache a l'index de la ligne de cache désignée, et permet de vérifier que le bloc chargé dans la ligne de cache correspond bien au bloc demandé par la requête mémoire. A partir de ces informations, la politique de placement intégrée dans le contrôleur de cache détermine la gestion interne des données.



(a) Mémoire cache avec une politique *direct-mapped*.

(b) Mémoire cache avec une politique *set-associative*.

FIGURE 1.5 – Exemple d'un cache avec une politique de placement dite *direct-mapped*. Source : Wikimedia Commons

Les Figures 1.5a et 1.5b présentent, à titre d'exemple, deux politiques de placement de caches différentes : une politique de placement dite *direct-mapped* et une politique de placement dite *set associative*.

Une politique de placement direct-mapped partitionne la mémoire principale en plusieurs blocs, et les associe chacune à une ligne de cache par correspondance modulo. Chaque bloc de données de la mémoire principale ne peut être stocké que dans une seule ligne de cache, mais chaque ligne de cache peut stocker plusieurs blocs de la mémoire principale. Lors d'un cache miss, le bloc de données couramment chargé dans la ligne de cache est réécrite en mémoire, avant de charger le bloc de données contenant le mot de données requis. On parle alors d'*éviction de ligne de cache*.

Dans le cas d'une politique de placement set-associative, l'ensemble des lignes de la mémoire cache est partitionné en plusieurs sous-ensembles, ou *sets*. Ce nombre de sous-ensembles appelé *degré d'associativité* du cache. La mémoire cache présentée a la Figure 1.5b possède, par exemple,

un degré d'associativité égal a 2, tandis qu'une mémoire cache avec une politique de placement direct-mapped possède un degré d'associativité égal à 1. Chaque bloc de données de la mémoire principale peut être stockée dans un nombre de lignes de cache égal au degré d'associativité de la mémoire cache. Lors d'un cache miss, le contrôleur de cache sélectionne parmi tous les sets une ligne de cache associée au bloc requis. Dans le cas où toutes ces lignes sont occupées, le contrôleur de cache procède alors à une éviction de ligne de cache selon des politiques qui varient en fonction de son implémentation. Ces politiques d'éviction de ligne de cache peuvent aller de l'élection aléatoire jusqu'à la sélection de la la ligne de cache la moins récemment utilisée.

Le choix des politiques de placement et d'éviction d'un contrôleur de la mémoire cache a un impact direct sur la latence des accès auprès de cette dernière, et les choix de conception d'une mémoire cache résultent souvent de considérations des cas d'usage logiciels et environnementaux.

Les mémoires ne sont pas sans inconvénients malgré cela. Leur mécanisme de chargement de données par blocs peut entraîner de la pression inutile sur la bande passante de la mémoire sous-jacente au contrôleur réalisant la requête, par exemple lorsqu'une donnée avec une faible localité spatiale et/ou temporelle est requise. Plus critique, la taille limitée des mémoires caches et surtout leur politique de placement peuvent entraîner des situations de misses dites *de conflits*, lorsque la capacité du cache est atteinte et qu'un bloc de données à écrire requiert une ligne de cache déjà possédée par un autre bloc. Dans une telle situation, il sera alors nécessaire de relâcher des données du bloc précédemment stocké pour permettre au bloc entrant d'être écrit. Ce problème de misses de conflits est exacerbé lorsque plusieurs régions requérant les mêmes blocs font partie du jeu de données de l'application ciblée. Dans un tel cas, les misses de conflits peuvent posséder une localité spatiale théoriquement élevée, mais avec un contrôleur de cache qui n'est pas suffisamment bien dimensionné pour pouvoir en bénéficier. Il est bien entendu possible de corriger le dimensionnement du cache, en augmentant sa capacité ou en adoptant des politiques de placement plus complexes par exemple. Par exemple, la politique de placement de cache *set-associative* est une autre politique de placement qui décompose le stockage de la mémoire cache en plusieurs voies d'associativé pour que les données cachées soient moins sensibles au misses de conflit (Figure 1.5b). Ces solutions restent à adopter dans la limite des contraintes de performance de l'architecture de calcul ciblée, puisque des solutions plus complexes ou physiquement plus grandes auront un impact sur la latence d'accès du cache[Smi82].

Il se passe alors une situation contradictoire à l'objectif initial des mémoires caches : très souvent le développement d'applications dites *haute-performance* requiert des développeurs de prendre connaissance de l'existence de la hiérarchie mémoire et de son dimensionnement pour correctement allouer et placer les données en mémoire, afin de maximiser autant que possible l'utilisation de la bande passante de l'architecture. Cette tâche peut s'avérer compliquée, du fait de la complexité des mécanismes en œuvre au sein des contrôleurs de cache et le manque d'informations fournies par les constructeurs industriels.

Pour ces raisons, les mémoires dites *scratchpads* sont également utilisées afin de fournir au logiciel un cache matériel programmable dans lequel organiser les données. L'avantage est de pouvoir mieux maîtriser le maintien des données au niveau des applications, mais l'utilisation efficace de scratchpads requiert une bonne connaissance du matériel ainsi qu'un profilage des applications ciblées pour identifier les données pertinentes à stocker sur ceux-ci. Il y a donc entre les mémoires caches et les mémoires scratchpads un compromis en termes de simplicité d'utilisation logicielle et d'efficacité.

1.2.1 Mécanismes de dissimulation de la latence d'accès à la mémoire

Plusieurs stratégies ont été proposées afin de masquer davantage la latence d'accès à la hiérarchie mémoire au sein des architectures de von Neumann. Les premiers mécanismes de ce genre mis en place, tel que les *stream buffers* reposaient sur les principes de localité spatiale et temporelle susmentionnés.

Un contrôleur de cache interfacé à un stream buffer effectue toutes ses requêtes auprès de ce dernier, qui va alors charger le bloc d'intérêt ainsi qu'une certaine quantité de blocs contigus à ce dernier dans une *file de préchargement*. À la réception du premier bloc, le contrôleur mémoire est libéré et peut fournir au CPU la reprise de son flot d'exécution, tandis que les blocs contigus sont chargés dans le stream buffer de façon non-bloquante. Grâce à ce mécanisme, il est possible de masquer efficacement les transferts de données disposant d'une très forte localité spatiale. Afin d'améliorer ce mécanisme, il est alors possible de dimensionner l'architecture d'un stream buffer selon certains attributs tels que le nombre de files et leur profondeur. Un tel dimensionnement matériel est bien sûr pertinent à la suite d'un profilage de performance sur une sélection d'applications, selon des contraintes matérielles de surcoût matériel établies.

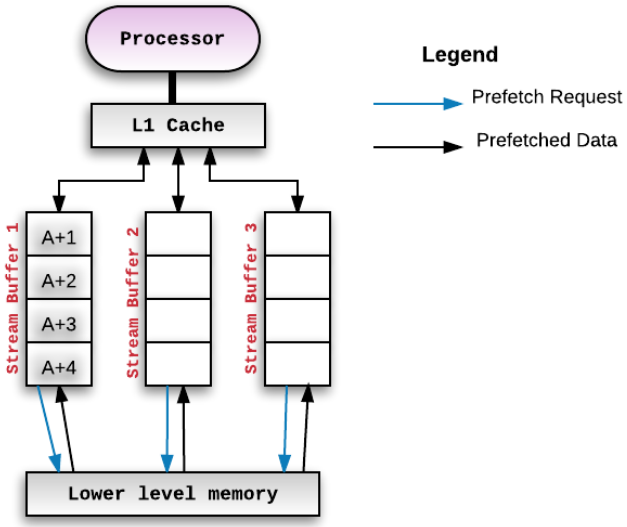


FIGURE 1.6 – Mécanisme de stream buffering à trois files, selon [Jou90].

Dans l'exemple présenté en Figure 1.6, le mécanisme de stream buffering proposé est dimensionné avec trois files chacune de profondeur 4. Bien que les stream buffers présentent un gain de performance important, ils ne fonctionnent à leur plein potentiel, de par leur conception, que sur des données stockées dans des blocs contigus. Lors du transfert de données présentant une forte localité spatiale mais ne présentant pas cette caractéristique de contiguïté, des transferts de blocs inutiles peuvent être générés, qui ce qui dégraderait considérablement la performance en termes d'efficacité énergétique par l'Energy Wall, et potentiellement en termes de temps d'exécution par l'introduction de misses de conflits supplémentaires.

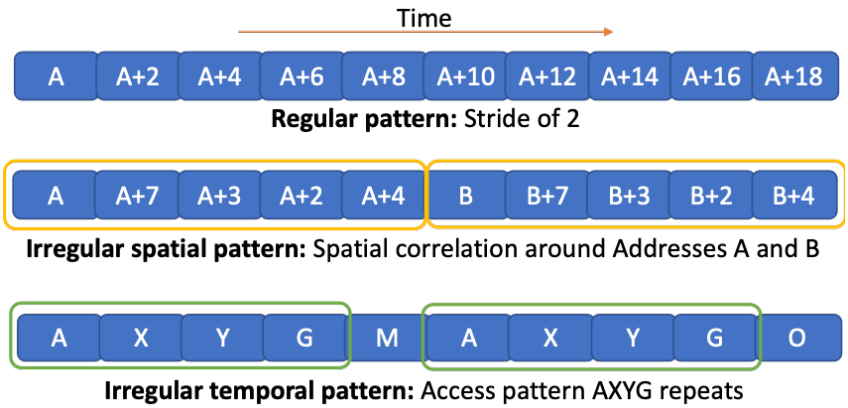


FIGURE 1.7 – Motifs d'accès à la mémoire détectés par différents types de mécanismes de prefetching. Image tirée de [Arm].

Pour ces raisons, des mécanismes dits de *prefetching* ont été élaborés et proposés dans l'art antérieur relatif à l'optimisation matérielle des contrôleurs de cache[VL00][Mit16][VL96][Ore00]. De façon générale, ces prefetchers surveillent les requêtes d'accès effectués par le contrôleur de cache afin de détecter des motifs d'accès à la mémoire et, selon un certain seuil de tolérance, déclencher le préchargement automatique de blocs de données en relation au motif d'accès détectés (Fig. 1.7). Les premiers prefetchers historiquement développés étaient les *stride prefetchers*, qui héritaient directement du design général des stream buffers avec la possibilité de détecter des lectures de blocs à des pas – des *strides* – réguliers. D'autres prefetchers se spécialisent dans la détection de motifs localement irréguliers, au sein d'une région mémoire donnée, mais globalement réguliers d'une région à l'autre. Ce genre de prefetchers sont très bons à efficacement précharger des blocs de données utiles en réponse à des accès à des structures de données complexes par exemple, là ou des stride prefetchers préchargeraient davantage de données inutiles et pouvant dégrader la performance d'exécution. Enfin, certains prefetchers sont également capables de précharger des données corrélées non pas par un motif d'accès spatial mais un motif d'accès temporel.

Par la nature heuristique de leur mécanisme, les prefetchers peuvent néanmoins dégrader les performances d'exécution et d'efficacité énergétique de leur hiérarchie mémoire hôte, pour des raisons similaires à celles énoncées lors de la présentation des stream buffers. De plus, les dimensions des motifs d'accès détectables par ces derniers, ainsi que leurs seuils de mise en action, ont un impact conséquent sur la mise à l'échelle de leurs implémentations physiques. Une solution proposée par certaines architectures est d'implémenter dans l'ISA des instructions de chargement de données non bloquantes afin de fournir aux utilisateurs les moyens d'implémenter du prefetching au niveau logiciel. Ces instructions sont souvent déclarées comme instruction de *software prefetching*, et utilisées par les compilateurs de code ou les utilisateurs à l'aide de directives ou de fonctions *intrinsèques*, des fonctions dédiées à paramétrer le fonctionnement du compilateur sur certains blocs de code.

1.3 Paradigmes alternatifs au modèle de von Neumann

1.3.1 Mémoires physiquement distribuées et topologies alternatives

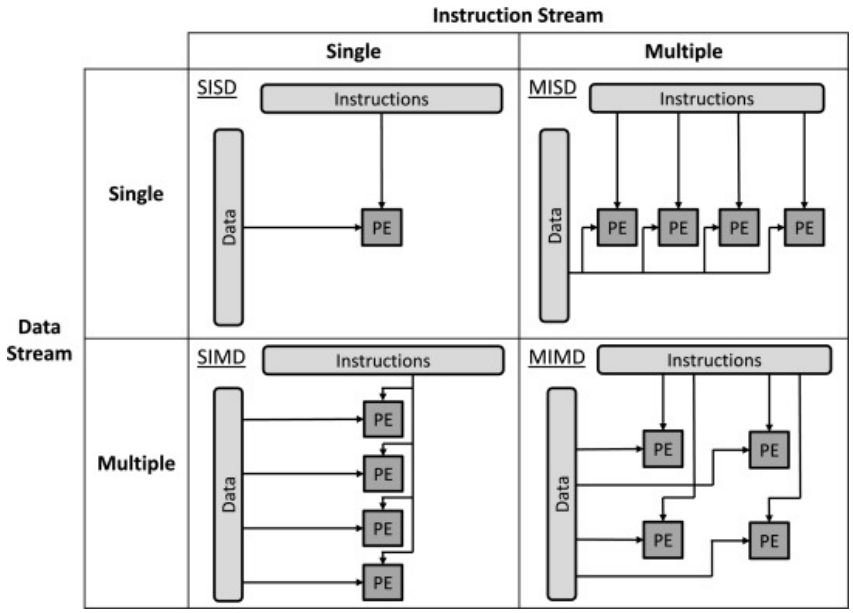


FIGURE 1.8 – Représentation des organisations architecturales selon la taxonomie de Flynn. Image tirée de [IRC11].

Comme discuté précédemment, l'écart de latence de fonctionnement entre un CPU et sa hiérarchie mémoire transforme cette dernière en goulot d'étranglement pour la performance des applications. Cependant, l'évolution de la performance des CPUs s'est également caractérisée par l'apparition des architectures multicœurs, dont le but est d'obtenir le support matériel de multiprocessus. De nombreuses solutions proposèrent des architectures multicœurs, afin d'augmenter le parallélisme d'exécution global des applications. Les cœurs de calcul furent également améliorés jusque dans leurs jeux d'instruction, avec l'intégration d'extensions spécialisés pour le calcul vectoriel tels que'Intel SSE/AVX, Arm NEON, et AltiVec pour les architectures PowerPC[UPD+20]. Ces extensions de jeux d'instructions, dites Single Instruction, Multiple Data (SIMD), et les architectures processeurs au modèle d'exécution *Multiple Instruction, Multiple Data* apportent de tous nouveaux degrés de parallélisme alors inexistants sur des machines monocœur. De nombreuses taxonomies furent rapidement proposées quant à la distribution du parallélisme au sein des architectures de calcul, l'une des premières étant la taxonomie de Flynn[Fly66]. Cette taxonomie, imagée en Figure 1.8, classifie les architectures de calcul selon l'organisation des *Processing Element*s (PEs) en fonction du flot d'instruction et de données :

— *Single Instruction, Single Data* (SISD) : correspond à un système mono-cœur tel que l'EDVAC précédemment présenté

— *Single Instruction, Multiple Data* (SIMD) (SIMD) : l'architecture contient un ensemble de PEs strictement synchrones qui partagent le même *Program Counter*, mais traite des

ensembles de données distincts. Les architectures SIMD correspondent par exemple aux architectures vectorielles et les extensions multimédias pour CPU précédemment mentionnées.

— *Multiple Instruction, Single Data* (MISD) (MISD) : le papier faisant suite à l'article introductif de la taxonomie de Flynn[FR96] présente ces architectures comme l'application d'un pipeline d'opérations indépendantes sur une même donnée. On parle également d'architectures *dataflow*, ou encore *systoliques*. L'idée d'architectures qui implémente un flot de calcul physiquement distribué n'est pas récente et remonte à des machines de calcul pionnières telles que le *Colossus Mark I*[Cop04]. Ce type d'architecture sera ensuite prisé dans des domaines spécifiques tes que le traitement du signal[FW87][SW83] et récemment l'Intelligence Artificielle (IA)[LWZC17].

— *Multiple Instruction, Multiple Data* (MIMD) : les architectures MIMD contiennent un ensemble de PEs totalement découplés, qui peuvent exécuter des flots d'instructions indépendants et sur des données indépendantes. Cela correspond aux architectures multicœurs.

Par l'émergence de ces degrés de parallélisme, de nouveaux besoins concernant la hiérarchisation et la distribution de la mémoire ont ainsi émergé.

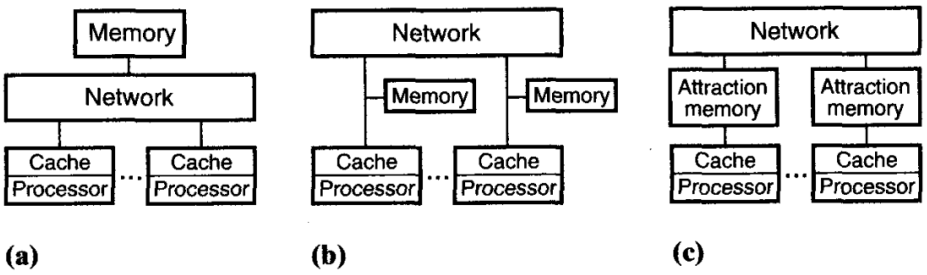


FIGURE 1.9 – Différents modèles d'architectures à mémoire partagée. (a) : Unified Memory Architecture, (b) : Non-Unified Memory Architecture, (c) : Cache-Only Memory Architecture. Image tirée de [HLH92].

Les premières architectures de calcul à mémoire partagée étaient très souvent basées sur le modèle *Unified Memory Architecture* (UMA), selon lequel tous les processeurs de l'architecture peuvent accéder à une même mémoire partagée avec une latence d'accès uniforme, c-à-d identique pour chaque processeur. L'avantage des architectures UMA est de grandement simplifier l'intégration de mémoire partagée à l'architecture, qui peut donc être utilisée simplement pour implémenter des routines logicielles d'*Inter-Process Communication* (IPC). Cependant, les architectures UMA présentent également le défaut d'être logiquement ainsi que physiquement implémentés par une seule unité de mémoire principale. Pour la majorité des interconnects réalisables dans une telle situation, la mémoire ainsi que le bus d'interconnexions deviennent des points de contention pour les processeurs de l'architecture, contention qui est aggravée au fur et à mesure que le nombre de processeurs augmente. De plus, le nombre de processeurs pouvant être implémenté sur de telles architectures mémoires est limité par des contraintes physiques liées au bus, ce qui limite le passage à l'échelle l'ajout de processeurs supplémentaires. Une solution matérielle pour faire passer à l'échelle les nombres de processeurs en relation avec la

mémoire partagée principale est alors de physiquement séparer cette dernière en plusieurs unités mémoires.

De ce partitionnement, deux grandes catégories d'architectures mémoires sont réalisables : les architectures à mémoire dites distribuées les architectures dites *Distributed Shared Memory* (DSM).

Dans le cas des architectures distribuées chaque processeur et sa mémoire principale privée composent un *nœud de calcul*. Pour pouvoir échanger des données entre eux, les nœuds de calcul disposent de routines logicielles de transfert de paquets, épaulées par des mécanismes matériels d'IPC.

Dans le cas des architectures DSM, plusieurs solutions peuvent être mis en place.

Une première solution est de permettre à chaque processeur d'accéder à l'ensemble de la DSM. Dans le cas où une donnée requise par un processeur donné se trouve dans la mémoire physiquement locale à ce dernier, la donnée sera garantie d'être chargée avec une pénalité d'accès moindre, comparé à un accès vers une mémoire physiquement distante. Ces architectures sont dites *Non-Unified Memory Architecture* (NUMA). Afin d'améliorer la performance de ces architectures, des mécanismes de cohérence de cache élaborés sont mis en place afin de limiter au maximum les accès à la mémoire tout en maintenant l'intégrité des données. Ces architectures sont nommées *Cache-Coherent NUMA* (cc-NUMA).

Une seconde solution associe à chaque processeur une mémoire principale qui est alors implémentée comme une mémoire cache. Dans le cas où une donnée requise par un processeur donné n'est pas présente dans sa mémoire principale physique, une requête de mise à jour et de cohérence de cache est effectué auprès des autres nœuds. Les architectures employant cette topologie et cette stratégie de cohérence de la mémoire sont dites *Cache-Only Memory Architecture* (COMA). Contrairement aux architectures NUMA, les architectures COMA ne considèrent pas qu'une donnée possède un habitat mémoire spécifique. Les données sont constamment déplacées d'un nœud à l'autre pour limite la redondance de données au sein de l'architecture. Ce mécanisme de transfert est cependant coûteux en termes de complexité.

Les modèles mémoires précédemment présentés ne sont pas exclusifs les uns des autres et peuvent être implémentés de concert au sein d'une même architecture de façon hiérarchique, à différents niveaux.

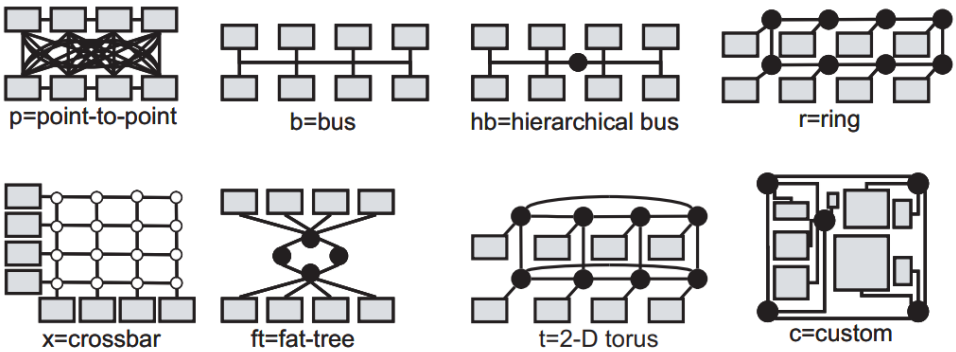


FIGURE 1.10 – Différents types de topologies de communication pour les Systems-on-Chip (SoCs). Comparaison tirée de [SKH08].

Toutes ces architectures mémoires ont également résulté en beaucoup d'efforts et de solutions concernant les topologies d'interconnexion (*interconnects*) entre nœuds de calcul [ZKCS02] [BM06] [SKH08] [KD17]. Si, auparavant des systèmes monocœur pouvaient se suffire d'un bus de conception dédiée, et les premiers multiprocesseurs d'un bus partagé, les architectures distribuées et DSM requièrent des topologies de bus plus avancées afin d'assurer le passage à l'échelle de systèmes massivement parallèles et multicœurs.

1.3.2 Technologies d'empilement 3D pour les mémoires et les architectures de calcul

L'empilement 3D est un paradigme de conception de circuits émergent, qui consiste à empiler physiquement des circuits intégrés afin qu'ils bénéficient d'une plus grande bande passante tout en déportant la surface effective du circuit sur sa hauteur. Ces circuits intégrés deviennent au sein de ce paradigme des successions de *plans 2D*, qui peuvent être conçus selon des techniques traditionnelles ou en utilisant elles-mêmes le paradigme de la conception 3D.

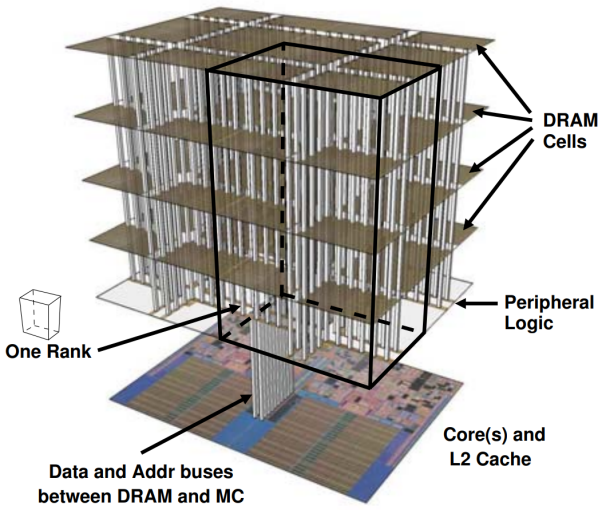


FIGURE 1.11 – Intégration d'une architecture multicœur 2D au sein d'une architecture multi-plans 3D. Les *Through-Silicon Vias* (TSVs) permettent de connecter les plans entre eux à de multiples endroits pour obtenir une large bande passante[Loh08].

La Figure 1.11 présente un exemple d'architecture a empilement 3D[Loh08]. La pile la plus basse est une architecture multicœur conçue selon les techniques traditionnelles 2D, et elle correspond dans ce contexte au plan de calcul de l'architecture. Ce plan de calcul va être connecté par le biais de Through-Silicon Vias (TSVs), des connecteurs entre plans 2D, à un empilement de mémoires DRAM à travers une première couche périphérique, correspondant au contrôleur de l'empilement de mémoires DRAM. Grâce à l'empilement 3D, une architecture de calcul 2D bénéficie d'une plus large bande passante que peuvent ne permettre les technologies de bus 2D classiques. Dans l'exemple présenté ci-dessus, la technologie de mémoire volatile DRAM

est employée, mais il est également possible d'utiliser d'autres technologies telles que le NAND FLASH[God21] ou encore des mémoires non-volatiles résistives RRAM[EBG+20].

L'empilement 3D reste cependant une technique de conception émergente, qui requiert encore du développement dans de nombreux domaines – Conception Assistée par Ordinateur, débogage, intégration système – pour être employée de façon efficace à une échelle industrielle. Elle est malgré cela une solution très prometteuse pour la conception de systèmes de calcul émergents.

1.3.3 Architectures de calcul hétérogène et paradigmes émergents

Afin de remédier aux limitations de performance des CPUs sur certaines applications spécialisées, certaines architectures de calcul intègrent une variété de nœuds de calcul afin d'augmenter la couverture d'applications sur laquelle la performance la plus proche de l'optimum puisse être atteinte. Nous présentons dans cette sous-section quelques-uns de ces nœuds de calcul intégrés couramment dans les architectures de calcul hétérogène.

Les ***Application-Specific Integrated Circuits*** **(ASICs)** sont des circuits spécialisés pour une seule et unique tâche, telle que le chiffrement et déchiffrement de clés cryptographiques, le rendu graphique de scènes 3D par *ray-tracing*, ou encore le calcul d'inférence pour des réseaux de neurones artificiels. Par leur spécialisation, les ASICs ont une portée de traitement informatique plus restreinte et ne sont pas garantis de supporter de futures applications ou traitement de tâches émergeant à l'avenir. De plus le temps de développement des ASICs est beaucoup plus long que celui des processeurs généralistes, puisqu'on recherche une efficacité optimale afin de rentabiliser le coût de développement d'un tel circuit.

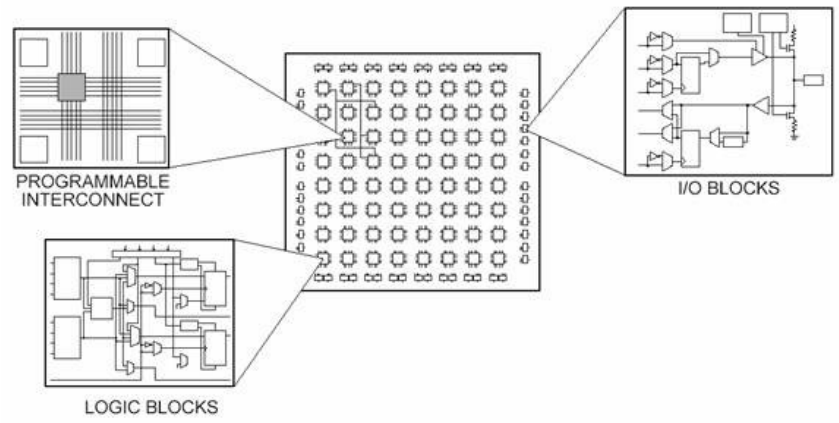


FIGURE 1.12 – Composition générale d'un *Field-Programmable Gate Array* (FPGA). Image tirée de [Par13].

Les ***Field-Programmable Gate Arrays*** **(FPGAs)** sont des circuits qui représentent une solution intermédiaire, car elles offrent une flexibilité applicative plus grande que les ASICs. En effet, les FPGAs peuvent être reprogrammés afin d'implémenter matériellement des descriptions de processus matérielles. La logique de reconfiguration des FPGAs entraîne pour une application donnée un surcoût de performance d'exécution comparé à une implémentation ASIC, mais qui

est bien plus performante qu'une implémentation CPU et qui permet d'adapter le FPGA à de futures applications.

Des circuits domaine spécifiques tels que les ***Digital Signal Processors* (DSPs)** peuvent également être intégrés pour implémenter matériellement des instructions généralistes à des applications telles que le traitement d'image ou du signal. On retrouve d'ailleurs dans une majorité de FPGAs des DSPs intégrés, faisant de ces nœuds de calcul des micro-architectures hétérogènes à part entière.

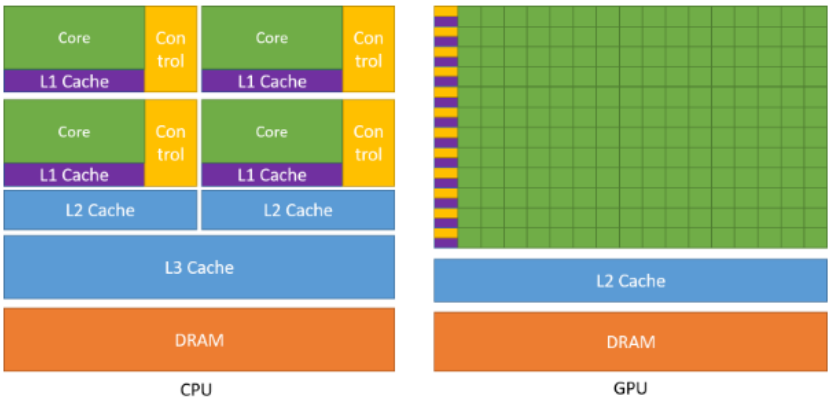


FIGURE 1.13 – Comparaison d'un *Graphics Processing Unit* (GPU) et d'un CPU. Source : Nvidia Corporation.

Les ***Graphics Processing Units* (GPUs)** sont des composants initialement conçus pour exécuter des pipelines de traitement graphique, mais ils sont aujourd'hui utilisés pour le calcul massivement parallèle généraliste, via l'intégration de multiprocesseurs. Ces multiprocesseurs sont orientés pour du calcul brut et avec très peu de contrôle du flot d'instructions, contrairement aux CPUs généralistes qui peuvent exécuter des codes aux flots d'exécution plus complexes pour moins de pénalités de performance. Le modèle architectural employé pour implémenter le parallélisme de ces multiprocesseurs a évolué au fil du temps, en passant par le SIMD et le MIMD[OLG+07]. Les architectures SIMD présentent l'avantage d'implémenter le support matériel du calcul parallèle avec un coût d'implémentation réduit, mais offre des performances moindres pour des applications qui génèrent des flots d'exécution complexes, tels que les codes conditionnels. Les architectures MIMD, quant à elles, intègrent plusieurs PEs dont le flot d'exécution est autonome, ce qui permet d'exécuter du code conditionnel avec efficacité. Cependant, le surcoût matériel pour l'implémentation d'architectures MIMD est beaucoup plus élevé. Aujourd'hui, la majorité des GPUs implémentent le modèle architectural dit *Single Instruction, Multiple Threads*, une sorte de *SIMD multi-threadé* qui compromet entre les deux précédentes approches[LNOM08]. Chaque multiprocesseur physique correspond à un fil d'exécution (*thread*) logique, et ces threads sont organisés en bloc. Ainsi, le modèle d'exécution d'un bloc de threads est perçu comme SIMD, mais chaque fil au sein d'un thread peut exécuter son propre fil d'exécution, à condition qu'il attende ses collègues de bloc pour se rejoindre à la fin de l'embranchement d'exécution (on parle là d'exécution en *lock-step*). Il est ainsi possible d'exécuter efficacement du code conditionnel au niveau matériel sans payer le plein coût d'implémentation du paradigme

architectural MIMD.

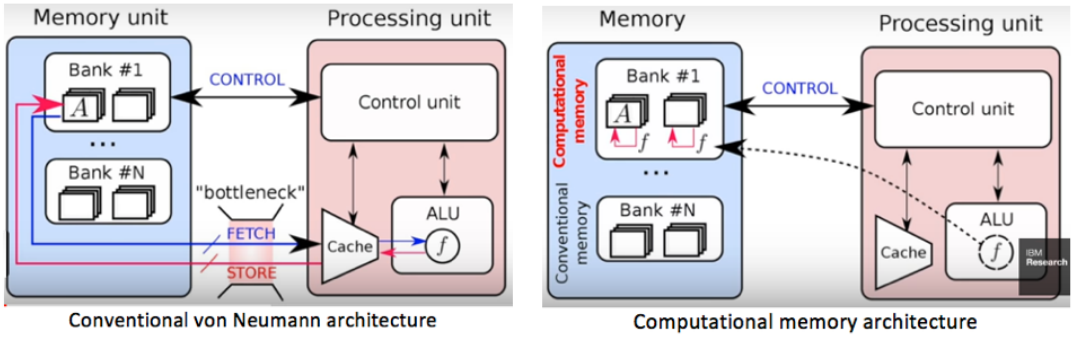


FIGURE 1.14 – Comparaison d'une architecture von Neumann classique et d'une architecture intégrant l'*In-Memory Computing* (IMC).

L'***In-Memory Computing*** **(IMC)** est un paradigme architectural et technologique qui résout les problèmes de goulot d'étranglement au sein des mémoires en y intégrant des capacités de calcul, très souvent parallèles. Par cette amélioration, l'échange de données entre le CPU et la mémoire cible est substitué par l'envoi d'un nombre inférieur de commandes de calcul auprès de cette dernière. Le résultat est donc une réduction de la consommation de la bande passante et, par extension, une meilleure efficacité énergétique. Il existe de nombreux moyens de concevoir des unités d'IMC, soit en réutilisant et en modifiant un circuit référence de cellules mémoire pour leur permettre de faire du calcul sur des données en plus de les stocker, soit en intégrant au sein d'une unité mémoire de référence des éléments de calcul, en périphérie des cellules mémoires.

Pour que l'IMC reste efficace, certains compromis de dimensionnement et de conception doivent être faits sur les éléments de calcul intégrés. Ces compromis ont un impact immédiat sur les dimensions et la précision des opérateurs arithmétiques supportés. Ces problèmes ne sont cependant plus aussi critiques qu'ils ne l'étaient auparavant pour une partie d'applications pour lesquelles le degré de précision requis a été substitué par une importante quantité de données. Bien que ces problématiques applicatives soient discutées dans le prochain chapitre, citons cependant les réseaux de neurones artificiels qui, grâce à la quantification des données, peuvent garder une précision de résolution remarquable tout en baissant la précision des opérations de calcul requis à l'implémentation.

Conclusion

En conclusion le modèle von Neumann, bien que souvent employé comme base pour des organisations d'architectures de calcul, présentent des limitations de performance liées à la mémoire principale par effet de bottleneck. Ces limitations, qui montrent un impact global sur la performance énergétique ainsi que l'efficacité des architectures von Neumann, ont poussé à terme les acteurs des domaines de l'électronique et de l'informatique à explorer de nouveaux paradigmes. Ces paradigmes se sont principalement manifestés par l'émergence d'architectures de mémoire globale distribuées, et par la démocratisation du calcul hétérogène. Plusieurs noeuds de calcul hétérogènes ont été présentés. Les ASICs, qui sont conçus et spécialisés pour des tâches

définies. Les circuits domaine-spécifique tels que les DSPs pour implémenter un plus large spectre d'opérations, les FPGAs dont la logique reconfigurable apporte un compromis entre flexibilité d'application et efficacité matérielle. Les GPUs qui offrent un degré de parallélisme massif. Et enfin l'IMC qui permet de réadapter les mémoires existantes pour faire du calcul.

Dans le cadre de ce sujet de thèse, le nœud de calcul étudié sera une architecture IMC, dont les spécifications seront données dans un chapitre suivant.

Chapitre

2 Défis logiciels pour la programmation des architectures non-von Neumann

- *Quels sont les applications d'intérêt existantes ?*
- *Quelles sont les solutions existantes pour décrire à haut-niveau ces applications et programmer les systèmes de calcul ?*

Introduction

Dans le précédent chapitre, nous avons conclu que les architectures de calcul ont évolué au-delà du modèle architectural traditionnel de von Neumann vers l'intégration de nœuds de calcul de plus en plus hétérogène. Rappelons que l'hétérogénéité d'une architecture de calcul se caractérise par la variété de conception et de spécialisation des nœuds de calcul. L'objectif des architectures hétérogènes et de couvrir une plage d'applications suffisamment large sur lesquelles l'architecture offre une performance la plus proche possible de l'optimum.

L'objectif de ce chapitre est de présenter les défis scientifiques liés à la programmation d'architectures de calcul hétérogènes et émergentes.

2.1 Expression des applications

L'origine derrière la conception de langages de programmation est la nécessité pour les programmeurs de pouvoir exprimer des problèmes à résoudre depuis un format d'expression compréhensible vers un dialecte propre à la machine visée pour exécution. Cette problématique de la programmation de machines peut déjà se retrouver dans la programmation des métiers Jacquard au début du XIX^{e} siècle[FJT12]. En effet, l'utilisation de cartes perforées pour programmer les motifs désirés dans les métiers à tisser naît de la nécessité d'encoder un motif à faire dessiner par le métier dans un format compréhensible des utilisateurs. Ce format des cartes perforées servira d'inspiration à Charles Babbage pour la programmation de sa *Machine Analytique*[Bro82] et elles seront une technologie employée pour le stockage de programmes et de données jusqu'à la fin des années 70, avant d'être remplacées par d'autres technologies de stockage. Avec le développement de l'informatique, de plus en plus de langages ont été créés pour apporter aux programmeurs une productivité satisfaisante dans la résolution de problèmes spécifiques à certains domaines.

Ainsi, le langage C était initialement conçu pour la programmation du système d'exploitation UNIX. La philosophie derrière la conception du langage était d'offrir une syntaxe suffisamment expressive pour faciliter la programmation lors du développement d'UNIX, alors que le langage assembleur était encore l'interface de programmation fondamentale des développeurs.

À cette époque, d'autres langages développés avec des problématiques spécifiques tels que COBOL ou encore FORTRAN étaient déjà employés selon les besoins. Face à l'évolution des applications pouvant bénéficier de solutions informatiques et numériques, ce sont de nombreux langages de programmation qui ont été développés, étudiés et proposés. Face à l'intérêt grandissant de la communauté scientifique autour des langages de programmation, de nombreux acteurs se sont intéressés à l'élaboration d'une taxonomie des langages de programmation. La classification des modèles de programmation permet de mettre en lumière des attributs et des métriques qualitatives, et de comparer les différentes propositions scientifiques du domaine.

Nous citons, dans le cadre de la présente thèse, la taxonomie proposée par Peter Van Roy, qui utilise comme critère classificateur le *paradigme* des langages de programmation. Il y définit un paradigme comme *"une approche à la programmation d'une machine de calcul selon un ensemble de théories mathématiques ou de principes cohérents"*[VR^{+}09]. La Figure 2.1 présente ladite taxonomie.

Nous pouvons retenir de cette taxonomie trois grandes familles de paradigmes, qui se distingue selon leur modèle d'exécution :

— Les paradigmes à états définis tels que C, OCaml, ou encore les langages *dataflow*.

— Les paradigmes à états indéfinis tels que Haskell, Prolog ou Scheme.

— Les paradigmes à états non-déterministes. Les langages implémentant ces types de paradigmes incluent des langages tels que Haskell ou Esterel.

Une question que nous pourrions être amenés à se poser est alors : *quels sont les intérêts d'employer un paradigme de programmation par rapport à un autre* ?

De façon pragmatique, un paradigme ne peut pas être considéré comme meilleur qu'un autre. Il permet, selon l'ensemble de théories sur lequel il est basé, une meilleure expression d'une catégorie de problèmes donnés. Ainsi, des avantages couramment attribués aux langages

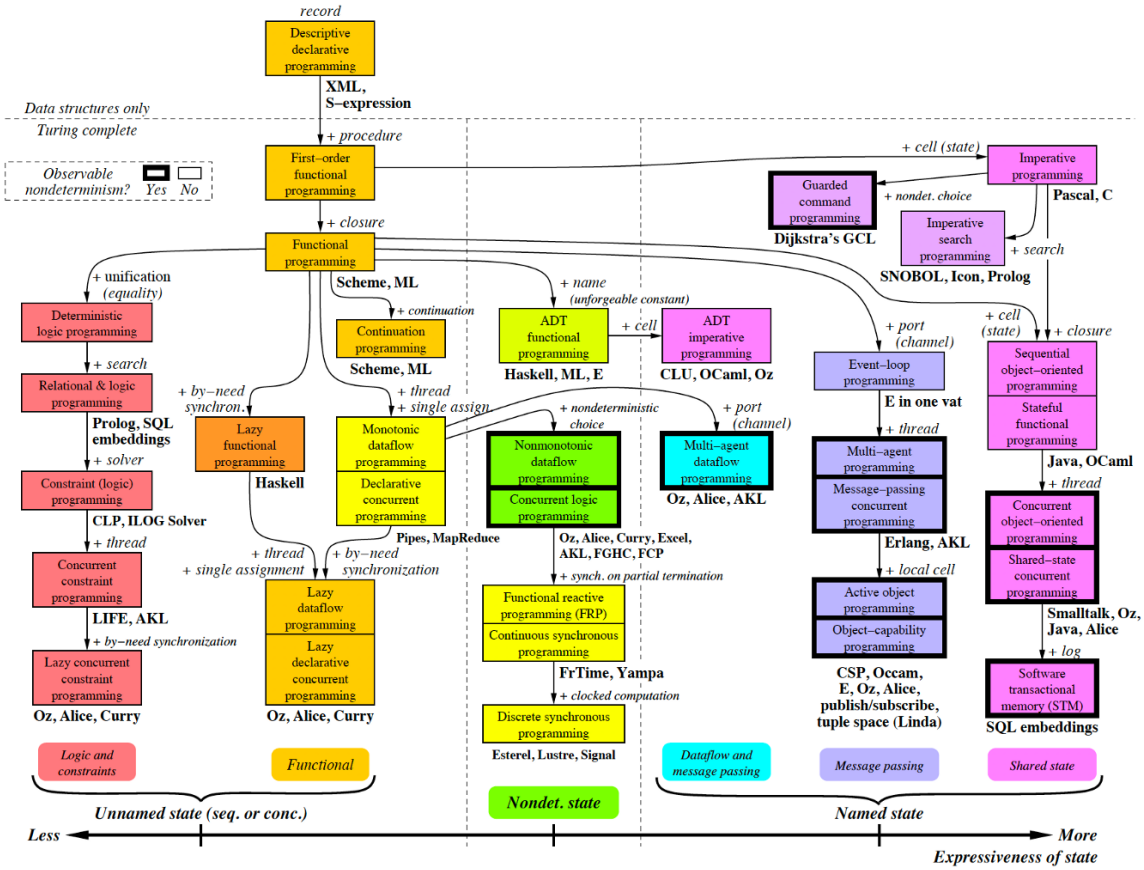


FIGURE 2.1 – Taxonomie des paradigmes des langages informatiques, tirée de [VR+09].

basés sur le paradigme fonctionnel sont la facilité de vérification et de test des programmes, là où un programme écrit dans un langage impératif, comme le C, est soumis à d'avantages d'aléas insolvables lors de la compilation de code.

En pratique, de nombreuses études empiriques concluent que le paradigme de programmation impérative reste le paradigme dominant dans le domaine industriel et professionnel [CDM+05] [KWL16], mais également dans le milieu de l'éducation [MR13] [KP05] [GK17].

Des langages tels que C et FORTRAN sont devenus ainsi des interfaces de calcul références pour la programmation de machines de calcul. Cependant ces langages, bien que généralistes, spécifient un modèle de programmation et d'exécution qui n'est pas comparable à celui supporté par des architectures matérielles tels que des architectures SIMD, multicœurs ou encore hétérogènes. Il a donc été nécessaire de développer des interfaces de programmation qui permettent d'accéder à ces fonctionnalités.

2.2 Portabilité de la performance des applications dans le contexte du calcul hétérogène

En conclusion, les langages sont l'interface fondamentale pour programmer des machines de calcul. Mais les langages généralistes, nativement, proposent peu voire aucune fonctionnalité syntaxique ou sémantique pour la programmation de nœuds de calcul hétérogènes, voire parallèles. Pour remédier à ce manque, différents types d'interface ont été proposés dans la littérature.

Catégorie	**Support syntaxique**	**Implem.**	**Exemples**	**Avantages / Inconvénients**
Librairies	Routines de programmation	Librairies et *packages*	— POSIX Threads, MPI — BLAS, LAPACK, FANN, OpenCV	[+] Syntaxiquement autonome [+] Modulaire [−] Inadapté à des applications non-standard [−] Divergence d'expressivité
DSLs	Langages spécialisés, extensions de langages	Compilateurs	— CUDA ISPC — OpenMP OpenACC OpenCL SyCL	[+] Syntaxe cohérente [+] Flexibilité infinie [−] Rétro-compatibilité limitée

Table 2.1 – Classification des modèles de programmation pour les architectures de calcul hétérogènes.

À la suite d'une étude bibliographique, nous dressons une classification des modèles de programmation pour les architectures de calcul multicœurs et hétérogènes. Cette classification, présentée en Table 2.1, utilise les critères suivants pour classifier les différentes catégories : le type de support syntaxique apporté par le modèle de programmation et le type d'implémentation employé. Nous discuterons, dans la suite de cette section, des différents avantages et inconvénients retenus pour chaque type d'interface.

2.2.1 Librairies et interfaces applicatives, ou *Application-Programming Interface* (API)

Les APIs sont des spécifications qui décrivent un ensemble de services et de routines. Ces routines peuvent être spécifiques à un domaine applicatif donné comme la modélisation phy-

sique, l'algèbre linéaire ou encore le calcul parallèle généraliste. Le développement et l'utilisation d'APIs proviennent d'un besoin de standardisation et d'uniformisation des applications. En effet, des applications appartenant au même domaine applicatif emploient souvent le même sous-ensemble d'opérations et de routines. Par exemple, la multiplication matricielle est une opération arithmétique très souvent utilisée dans des applications d'algèbre linéaire, et il est donc possible de déduire un intérêt à standardiser l'emploi de cette opération arithmétique au niveau logiciel. C'est par ce besoin de standardisation que des spécifications telles que BLAS[Sta01] et LAPACK[ABB+99] pour l'algèbre linéaire, ou encore OpenCV[Bra00] et OpenVX[Gro17] pour le domaine de la Computer Vision. Des méta-domaines tels que la programmation pour architectures parallèles et distribuées ont également bénéficié d'API standards pour uniformiser les applications. Ainsi, la spécification POSIX Threads (ou *pthreads*) est devenue la spécification *de facto* pour le support logiciel de la création de processus. Nous pouvons également mentionner la spécification *Message Passing Interface* (MPI) qui offre une interface de programmation pour la programmation et l'interaction de plusieurs machines (voire nœuds) de calcul au sein de systèmes distribués.

L'utilisation d'interfaces standards tels que les APIs pour le développement d'applications aux domaines spécifiques donne lieu à des conséquences positives. Ainsi, la standardisation des applications par le biais d'APIs rend les architectures de calcul émergentes plus faciles à adapter pour être supportées par des projets logiciels antécédents, dès l'instant où elles sont fournies avec une API qui implémente les services requis par les spécifications employées. Il est alors possible de parler de modularité des applications standards vis-à-vis des architectures de calcul. De plus, cette standardisation des interfaces permet de cacher *en boîte noire* des subtilités propres à l'implémentation des routines et des services fournis par l'architecture. Les APIs peuvent ainsi être implémentées sous la forme de librairies à invoquer depuis le code réquisitionnant les services, voire sous la forme de librairies dynamiquement liées, ou *Dynamically-Linked Libraries* (DLLs).

Il est cependant important de mitiger ces "points forts" et de préciser que les APIs possèdent des limitations intrinsèques. Rappelons ainsi que les spécifications sont dressées selon des analyses de bases de données de code et de tendances spécifiques à des domaines applicatifs données. Ces études sont empiriques et peuvent donner lieu à des spécifications dont le **surcout de conformité** peut donner lieu à des implémentations d'applications aux performances sous-optimales, comparées à une implémentation théorique parfaite pour une architecture donnée. Certains domaines applicatifs se sont vus agrémentés de nouvelles spécifications à la suite d'études a posteriori, suite à la découverte de telles limitations. C'est notamment le cas de BLIS, une spécification et un environnement de développement dédié à l'implémentation des librairies BLAS efficaces. Pour ce faire, BLIS définit une spécification de sous-ensembles de routines d'avantages partitionnées, qui permettent de réduire le surcoût de conformité tout en facilitant l'implémentation de librairies BLAS. BLIS est ainsi un exemple d'API nouvelle qui maintient la rétrocompatibilité des applications utilisant la spécification BLAS. Cependant, certaines APIs spécialisées peuvent se retrouver obsolètes après un certain temps, car dépassées par de nouvelles spécifications sur un ensemble de critères qualitatifs. C'est ainsi que l'API LINPACK s'est retrouvée obsolète suite au développement de LAPACK[ABB+99], qui implémentait davantage de fonctionnalités avec une plus grande rigueur dans la spécification des routines pour garantir une performance d'exécution améliorée, notamment avec l'utilisation efficace des mémoires

caches par le biais d'optimisation logicielle. Dans de telles situations, la ré-implémentation d'applications antécédentes est nécessaire. Enfin, l'utilisation d'APIs standardisées ne permet pas d'implémenter des applications qui n'emploient pas ces dits standards. Il semble donc naturel de conclure que l'utilisation d'APIs applicatives est insuffisante pour offrir le plein support d'architectures de calcul de façon générale.

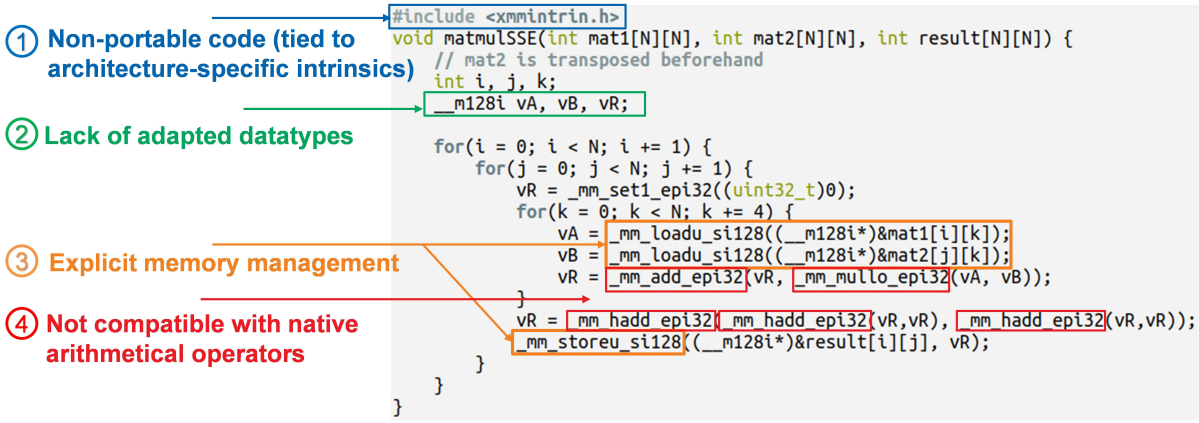


FIGURE 2.2 – L'implémentation de la multiplication matricielle avec l'API `xmmintrin.h` pour les architectures Intel SSE/AVX. Deux modèles d'exécution sont syntaxiquement présents dans le code (scalaire/SIMD).

Face à la problématique du support d'architectures de calcul émergentes, les APIs peuvent être employées comme interface entre le programmeur et des fonctionnalités fondamentales telles que le jeu d'instructions ou les opérations de transferts de données. Les fonctions servant à invoquer des instructions spécialisées sont appelées *instructions intrinsèques*. L'utilisation d'instructions intrinsèques entraîne alors l'intégration d'un second flot d'exécution qui est syntaxiquement distinct de celui offert par le langage de programmation hôte. Ce phénomène de ***divergence syntaxique*** rend la programmation difficile à gérer pour les programmeurs, et limite l'adoption d'architectures de calcul aux fonctionnalités émergentes.

Pour illustrer ce phénomène, nous présentons à la Figure 2.2 un code implémentant la multiplication matricielle pout les architectures Intel SSE/AVX. L'implémentation du code est réalisée sur la base du langage C, et de la librairie `xmmintrin.h`. Nous pouvons remarquer que, par manque explicite de support pour les architectures de calcul SIMD d'Intel, les ressources des extensions SIMD ne sont pas explicitement supportées au niveau des types de données, des opérateurs syntaxiques et des accès mémoires. Le résultat est la décomposition du code en deux flots d'exécutions sémantiques différents.

Ainsi, bien qu'il s'agisse d'une solution fonctionnelle, les APIs de support matériel ne sont pas suffisantes pour la programmation à haut niveau des architectures de calcul hétérogènes.

2.2.2 Langages de programmation dédiés, ou *Domain-Specific Languages* (DSLs)

Les langages dédiés ou *Domain-Specific Languages* (DSLs) sont des langages de programmation spécialisés pour un domaine applicatif précis. L'avantage des DSLs est de fournir, selon les besoins applicatifs, du support syntaxique permettant de faciliter l'implémentation d'applications selon les besoins spécifiés. À l'instar des APIs qui peuvent être applicatives ou de support matériel, les DSLs peuvent être spécifiées pour des domaines applicatives précis ou pour le support logiciel d'architectures – ou d'ensembles d'architectures – données. Afin de donner suite aux conclusions du paragraphe précédent, nous nous focaliserons dans la suite de cette section sur les DSLs et environnements de programmation dédiés pour les architectures de calcul hétérogènes.

```
__device__
void axpy(float a,
          float * restrict x,
          float * restrict y,
          int len)
{
  int i = blockIdx.x*blockDim.x
          + threadIdx.x;
  if (i < n)
    y[i] = a*x[i] + y[i];
}
```

(a) Implémentation en Nvidia CUDA C++.

```
export void axpy(uniform int A,
                 uniform int X[],
                 uniform int Y[],
                 uniform int len)
{
    foreach (i = 0 ... len)
    {
        int a = A;
        int x = X[i];
        int y = Y[i];
        y += a * x;
        Y[i] = y;
    }
}
```

(b) Implémentation en Intel ISPC.

FIGURE 2.3 – Implémentation de la routine `axpy` pour les architectures Nvidia et Intel.

Une première catégorie de DSLs pour la programmation des architectures de calcul est les DSLs dits *propriétaires*. Il s'agit de langages spécialisés pour la programmation d'un ensemble d'architectures données. Très souvent, ces DSLs sont développés par les entreprises propriétaires des propriétés intellectuelles attachées aux architectures cibles. Nous pouvons par exemple citer l'environnement CUDA, développé par Nvidia pour ses GPUs et qui fournit une interface de programmation C++. Cette interface introduit le concept d'*espaces d'exécution*, afin que les fonctions puissent être paramétrées pour être exécutées sur les différentes ressources globales de l'architecture. Dans l'exemple montré à la Figure 2.3a, le paramétrage de la routine `axpy` avec le mot-clé `__device__` indique la fonction est exécutée par le GPU et appelée depuis le HPU.

ISPC est un autre exemple d'Intel de DSL propriétaire, pour la programmation de ses architectures SIMD et de ses GPUs. ISPC utilise de façon implicite la même notion d'espace d'exécution, avec pour impact principal l'emploi des types de données hérités depuis les langages C/C++. Dans l'exemple présenté en Figure 2.3b, les variables de types int sont associées aux ressources SIMD, et la spécification de variables non vectorielles (tels que les paramètres de la fonction) s'effectue avec le mot-clé `uniform`.

Les deux modèles de programmation présentés sont des modèles de programmation dits *Single Program, Multiple Data* (SPMD). Le caractère parallèle du flot d'exécution est implicite au niveau de la syntaxe pour maintenir une cohérence syntaxique entre le langage hôte (pour la programmation du HPU) et le DSL.

Une limitation intrinsèque aux DSLs propriétaires est la limitation de leur support matériel. Un DSL propriétaire ne pourra supporter qu'un ensemble d'architectures, ce qui rend la portabilité vers d'autres architectures hétérogènes potentiellement difficile.

C'est ainsi que d'autres solutions furent développées pour permettre la programmation d'architectures de calcul parallèles et d'accélérateurs tout en assurant un degré de portabilité. Nous pouvons notamment mentionner OpenACC [WST+12] et OpenCL[Mun09], qui utilisent deux approches syntaxiques différentes pour résoudre cette problématique mais intègrent communément la notion de *pilotes*. Ces pilotes décrivent les opérations disponibles pour des architectures données et sont invoquées depuis l'extérieur du code, afin que ce dernier soit portable sur toutes les architectures. Le support de nouvelles architectures de calcul émergentes s'effectue alors par le développement de pilotes pour le modèle de programmation.

Il est cependant important de noter que bien que cette approche garantisse la portabilité de la fonctionnalité des applications, d'une architecture matérielle à une autre, elle ne garantit pas l'atteinte de la performance optimale pour une application donnée, à cause du surcoût d'exécution des environnements d'exécution pour la gestion de la distribution de calcul. Ce surcoût est donc dépendant des fonctionnalités et des choix de conception de chacun de ces DSLs de programmation hétérogène portable. En pratique, une interface de programmation peut implémenter – et très souvent pour les architectures hétérogènes, implémentent – de multiples sous-interfaces selon différentes philosophies pour offrir aux programmeurs le meilleur rapport entre productivité et couverture applicative. L'entreprise Nvidia, en pratique, fournit aux programmeurs un Toolkit complet qui intègre des extensions de langages, des APIs propriétaires ainsi que des APIs standard[NVI]. Intel effectue également le même exercice avec la maintenance d'ISPC, mais également d'autres outils logiciels comme l'Intel C++ Compiler et la *Math Kernel Library* (MKL). Ces outils, ainsi que d'autres interfaces de développement ont été récemment uniformisés sous le projet OneAPI, qui répond aux mêmes besoins que le CUDA Toolkit pour les produits du constructeur, mais également pour les accélérateurs grâce à l'intégration de *SYCL*, surcouche du projet OpenCL précédemment mentionné[Int].

2.3 Les stencils : une classe d'algorithmes commune

Dans un souci de mieux comprendre les besoins matériels des applications et ainsi établir une cartographie des besoins logiciels, de nombreuses études de tendance ont été réalisées pour proposer une classification des applications numériques.

Une étude qui s'est particulièrement démarquée est celle initialisée par Philipp Collela, qui sera reprise par l'Université de Berkeley et qui donnera lieu aux ***dwarfs***, un ensemble de méthodes algorithmiques couramment employées dans l'implémentation informatique de solutions numériques[ABC+06].

L'un de ces dwarfs est le #5, intitulé *structured grids*, qui est plus couramment connu sous le noms de *codes de stencils*. Un code de stencil est un algorithme qui applique sur tous les éléments

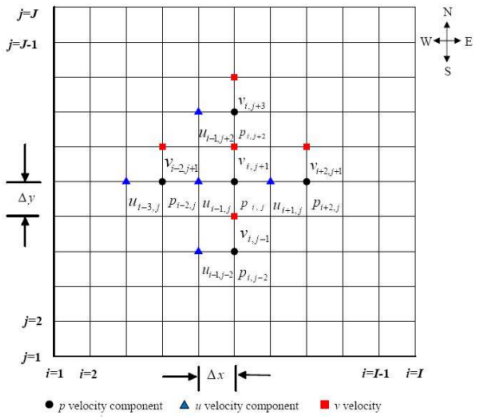
d'une grille structurée, une fonction de calcul dépendant pour chaque point d'un voisinage donné, relatif à sa position.

Cette classe d'algorithme peut être exprimée de façon formelle par la définition suivante[Sch10] :

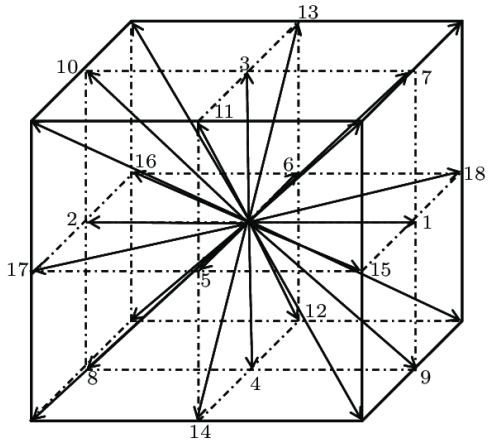
$$G = (V, E) \quad \text{(Grid)} \tag{2.1}$$
$$V \quad \text{(Set of cells in the grid)} \tag{2.2}$$
$$E \subseteq V \times V \quad \text{(Neighbourhood relationship)} \tag{2.3}$$
$$S \quad \text{(Set of states)} \tag{2.4}$$
$$u : V \times \mathcal{P}(V) \times (s : V \Rightarrow S) \Rightarrow S \quad \text{(State update function)} \tag{2.5}$$
$$s_0(v) \quad \text{(Initial state)} \tag{2.6}$$
$$s_{t+1}(v) = u(v, N_V, s_t) \quad \text{(Next state assignment)} \tag{2.7}$$

G correspond à la grille sur laquelle le code de stencil est appliqué, avec V les éléments de la grille et E la définition du voisinage employé par la fonction de calcul. s_0 est l'état initial de la grille et s_{t+1} l'état de la grille à l'instant t+1, c'est-à-dire après l'application de la fonction u sur s_t.

Les applications de stencils sont présentes des propriétés qui expliquent leur popularité dans des domaines tels que la modélisation de fluides, le traitement d'image ou encore l'algèbre linéaire.



(a) Jeu de stencils pour l'approximation aux équations de Navier-Stokes à deux dimensions, proposé par [RKA+11].



(b) Stencil pour le *Lattice Boltzmann Method* (LBM) à trois dimensions et dans 19 directions, dit D_3Q_{19}. Image tirée de [GFJ+20].

FIGURE 2.4 – Stencils employés dans des algorithmes de simulations de fluides newtoniens et/ou non-newtoniens.

Les codes de stencils s'appliquent sur des structures de données ordonnées à deux ou trois dimensions, qui disposent donc d'une forte localité spatiale et temporelle. Cet attribut de ce genre de structures de données est à l'avantage de techniques telles que la *résolution d'équations*

par différences finies, aussi connue sous le nom de *Finite Difference Methods* (FDMs)[GRS07]. Le principe fondamental des FDMs est la discrétisation d'équations différentielles afin qu'elles puissent être résolues par approximations successives. La Figure 2.4, par exemple, présente des stencils issus de la discrétisation d'équations servant à la modélisation de fluides. Ces codes de stencil sont ensuite employés dans des logiciels de modélisation et de simulation physique.

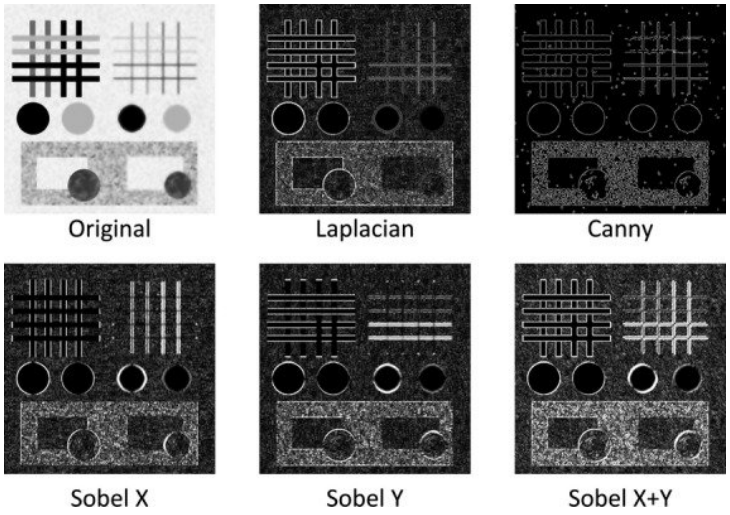


FIGURE 2.5 – Présentation de l'effet de différents filtres convolutifs pour la détection de contours, tirée de [Uch13].

Il est également possible d'employer la discrétisation d'équations différentielles en traitement d'image pour en extraire des attributs spécifiques. Ainsi, l'opérateur laplacien discret est employé en traitement d'image pour l'extraction de contours et la détection de mouvement sous la forme de convolutions[WDL20]. Il ne s'agit, bien sûr, pas du seul filtre pouvant être utilisé à cette fin, et d'autres filtres convolutifs ont été proposés dans l'art antérieur (Figure 2.5).

De façon plus générale, l'opération de convolution peut être vue comme un code de stencil particulier. Ainsi, des applications telles que la Computer Vision et le Machine Learning, par le biais des Réseaux de Neurones Convolutifs (CNNs), peuvent être implémentées sous la forme de codes de stencil.

Cette discrétisation d'équations différentielles potentiellement complexes à résoudre permet l'implémentation efficace de nombreuses applications numériques et scientifiques sur des architectures de calcul parallèle. En somme, les codes de stencil sont une classe d'algorithmes souvent employés.

Les codes de stencil impliquent, par construction, le transfert de beaucoup de données redondantes, dont les proportions varient selon la définition dudit code. Ainsi une itération de Jacobi, telle que l'implémentation C présentée à la Figure 2.6, définit un voisinage de stencil de cinq éléments et entraînera au cours de son exécution la consommation d'une quantité de données proportionnelles à cinq fois la taille de la structure de donnée manipulée. Cette quantité de données ajoute une pression importante sur la bande passante de la mémoire, qui peut être aggravée selon l'implémentation du code de stencil. L'implémentation de l'itération de Jacobi, précédemment présentée, est ainsi fonctionnelle mais entraîne des motifs d'accès mémoires

```
for (int t = 0; t < T; t++) {
  for (int i = 1; i < N-1; i++) {
    for (int j = 1; j < N-1; j++) {
      B[i][j] = 0.2 * (A[i][j]
                     + A[i][j-1]
                     + A[i][j+1]
                     + A[i-1][j]
                     + A[i+1][j]);
    }
  }
  // swap pointers
  temp = A; A = B; B = temp;
}
```

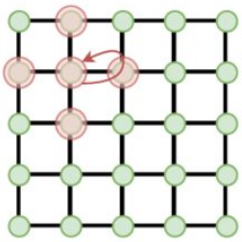


FIGURE 2.6 – Implémentation naïve d'une itération jacobienne, tirée de [DBH+21a]. Chaque élément de A est lu au moins 5 fois, par la taille du voisinage du stencil.

qui sont localement irréguliers, bien que la structure de données elle-même dispose d'une forte localité spatiale et temporelle.

Il est donc nécessaire de mettre en place une solution qui permette le calcul efficace de codes de stencil. La suite de la présente section va présenter l'art antérieur concernant les solutions proposées à cette fin. Deux axes d'approche seront abordés : un premier quant aux solutions d'accélération matérielle de codes de stencils, et un second concernant les solutions logicielles de programmation, et d'optimisation de cette catégorie d'algorithmes.

2.3.1 État de l'art du support matériel des calculs de stencils

Lors du précédent chapitre, nous avons discuté au développement croissant de nœuds de calculs émergents et d'accélérateurs dédiés. Il n'est donc pas étonnant de trouver dans l'art antérieur des propositions d'accélérateurs dédiés au calcul de stencils. Nous remarquons, à la suite d'une étude bibliographique dans le domaine, que la majorité des propositions architecturales se présentent sous la forme de soft-IPs dédiés à être synthétisées sur FPGA, et emploient des principes liés au calcul proche-mémoire. Ce deuxième point est sensé, du fait que la forte proximité et la maximisation de la bande passante accessible par les PEs permettent l'accélération de calculs de stencils.

Ainsi, des accélérateurs soft-IP tels que SODA[CCWZ18], NERO[SDH+20] ou encore NARMADA [SDH+19] proposent des architectures implémentées au sein d'un FPGA pour l'accélération de stencils, à l'aide de mémoires distribuées vers des PEs massivement parallèles. Ces solutions présentent des gains de performance cohérents avec les stratégies d'optimisation matérielles sur lesquelles elles se basent. Il faut cependant rappeler que la performance de ces architectures est liée à celles des FPGAs sur lesquels elles sont synthétisées. Ainsi, l'implémentation d'accélérateurs de stencils sur FPGA donne lieu à des performances moindres comparées à des implémentations ASIC en termes de temps d'exécution et de consommation énergétique[NSS+16][KR07][NSAR20]. Du fait de l'importance des codes de stencil dans les applications numériques, il semble donc plus avantageux d'implémenter de façon permanente des ressources matérielles pour l'accélération de codes de stencils.

D'autres solutions matérielles comme PIMS[LWT+19] ou CASPER[DBH+21b] prennent ce

parti pris sous la forme d'implémentations ASICs. Leurs micro-architectures intègrent de la logique permettant la réutilisation de données pour le calcul de stencils convolutifs. Nous notons cependant que cette restriction aux calculs convolutifs limite leur portée applicative, par rapport aux différentes applications précédemment mentionnées ainsi que le statut des codes de stencil dans la littérature des applications numériques. Il y a donc un intérêt particulier à permettre le calcul efficace de codes de stencil au sein d'architecture de calcul généralistes.

2.3.2 État de l'art du support logiciel des calculs de stencils

De nombreuses solutions et méthodologies logicielles ont été proposées pour l'implémentation efficace de stencils sur architectures CPU et GPU. Comparées aux implémentations ASICs mentionnées, ces architectures intègrent une bande passante globale ainsi qu'une hiérarchie mémoire bien plus large, mais pas de support matériel pour le calcul de stencils.

L'utilisation de paradigmes d'optimisations tels que le modèle polyédrique [GLW98] donne accès à la possibilité aux programmeurs d'implémenter des applications basées sur des codes de stencils en passant par le biais de DSLs implicitement optimisés et réordonnancées. Ainsi, des langages de programmation tels que Hallide[RKAS+17] et PolyMage[MVB15] permettent d'implémenter à haut-niveau des pipelines de traitement d'image. Nous pouvons également citer Darkroom[HBD+14], qui implémente son propre environnement d'optimisations et supporte au niveau de ses sorties des implémentations CPU comme FPGA, par le biais de la génération de modèles synthétisables.

Il existe également des méthodologies d'optimisation systématique des pipelines de traitement d'image tels que les *High-Level Transforms* [LEHZ+14], avec la particularité de pouvoir s'adapter facilement à des architectures de calcul multithreads et supportant des opérations SIMD.

D'autres solutions plus généralistes ont été proposées, à différentes échelles. Les compilateurs Polly[GGL12] et PoCC[PBB+09] permettent la transformation polyédrique de codes sources arbitraires de façon totalement indépendante des architectures de calcul. L'environnement d'optimisation MLIR[LAB+20] se base en partie sur ces travaux pour permettre l'optimisation efficace de calculs de réseaux de neurones. Nous pouvons également mentionner Tiramisu[BRR+19], qui propose un DSL pour diriger l'optimisation polyédrique de code.

La mention de Tiramisu nous permet d'introduire une problématique soulevée par l'avènement de l'hétérogénéité des architectures de calcul : le contrôle des optimisations de code. En effet, si l'ordonnancement de code pour maximiser la réutilisation des données peut se faire avec la seule utilisation d'analyse de code algorithmique, l'association du motif d'ordonnancement résolu sur une architecture de calcul donnée requiert que le compilateur ait conscience des ressources de calcul et de stockage de cette dernière. Il y a donc là une problématique quant à la portabilité de cette catégorie d'optimisations. Des projets logiciels tels que TVM[CMJ+18] proposent des systèmes de pilotes, à la manière de OpenCL pour paramétrer les optimisations de code. D'autres langages tels que RISE[SKKP22] prennent des approches radicalement tangentielles en intégrant au sein de l'environnement de programmation un DSL dédié à l'implémentation des applications et un second DSL pour la description des optimisations logicielles, ce qui permet de maintenir un certain degré de portabilité.

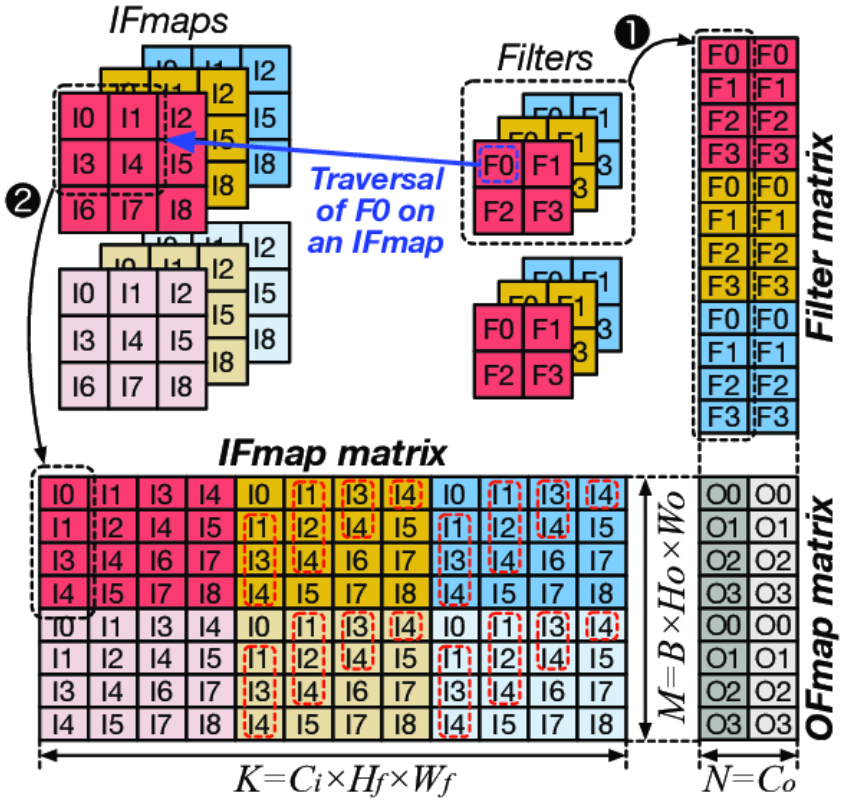


FIGURE 2.7 – Transformation de convolutions en im2col pour le calcul de CNNs. Image tirée de ([LLO+19]).

Certaines de ces solutions sont complémentées par la transformation des structures données sous un format permettant de maximiser l'utilisation de la bande passante, pour atteindre des vitesses d'exécution améliorées. L'*Image-to-Column* (*im2col*) est ainsi une transformation de structures données qui duplique et réorganise des jeux de filtres convolutifs sous forme matricielle. L'intérêt de cette transformation dans le cadre des CNNs est qu'elle permet de réaliser l'inférence des réseaux de neurones avec de l'algèbre linéaire dense, plus particulièrement la multiplication matricielle. Les routines d'algèbre linéaire présentent l'avantage d'être hautement optimisées et spécialisées pour l'intégralité des architectures CPU et GPU, grâce à leurs implémentations d'APIs telles que BLAS ou LAPACK (voir section précédente). La transformation im2col s'est ainsi imposée comme opération fondamentale pour le calcul des CNNs, et a donné lieu à d'autres propositions de transformations de filtres convolutifs pour une meilleure exploitation de la bande passante.

Cependant, ces transformations impliquent une expansion de données qui n'est pas négligeable. La Table 2.2, tirée présente une comparaison du surcoût en Kilo-octets de l'empreinte mémoire de filtres de convolutions transformés selon trois algorithmes. En fonction de la configuration (S, K, C_{in}, C_{out}) et de l'algorithme employé, ce surcoût peut être de l'ordre d'environ 1 Kilo-octet $(S = 8, K = 3, C_{in} = 8, C_{out} = 8)$ et peut s'élever à la trentaine de Méga-octets $(S = 100, K = 5, C_{in} = 128, C_{out} = 128)$. Cette augmentation de l'empreinte mémoire

		K	3		5	
S	$\mathbf{C_{in}}$	$\mathbf{C_{out}}$	8	128	8	128
20	8	A	8.44	8.44	23.44	23.44
		B	0.95	15.25	1.97	31.5
		C	22.78	22.78	50	50
	128	A	135	135	375	375
		B	0.95	15.25	1.97	31.5
		C	364.5	364.5	800	800
100	8	A	8.44	8.44	23.44	23.44
		B	4.7	75.25	9.47	151.5
		C	675.28	675.28	1760	1760
	128	A	135	135	375	375
		B	4.7	75.25	9.47	151.5
		C	10550	10550	28120	28120

TABLE 2.2 – Surcoût de Kilo-octets l'empreinte mémoire de filtres convolutifs selon les transformations *im2col*(A), *kr2row-aa*(B) et *p-im2col*(C). L'expérience considère l'application $C_{in} \times C_{out}$ filtres de taille $K \times K$ sur une image de taille $S \times S \times C_{in}$. Résultats tirés de [TLNA21].

peut poser problème en fonction des dimensions du système mémoire de l'architecture ciblée, notamment des architectures embarquées basse-consommation pour l'*Internet-of-Things* (IoT)[MPTC+20][SCR+17][GRC+20]. De plus, l'augmentation de l'empreinte mémoire ne pose pas seulement des problématiques de stockage mais également de consommation énergétique. En effet, si cette expansion des données permet une plus grande consommation de la bande passante et une plus grande vitesse d'exécution des convolutions, elle entraîne également une aggravation de l'Energy Wall, phénomène que nous avons présenté lors du précédent chapitre. Il faut également considérer le surcoût de la transformation à effectuer a priori de l'exécution de la convolution, bien que l'utilisation de fonctionnalités de la hiérarchie mémoire, notamment les mémoires caches, permette de mitiger ce surcoût. Cependant, une implémentation de transformation de données efficace requiert alors une méthodologie d'implémentation qui soit dimensionnée selon la hiérarchie mémoire de l'architecture ciblée pour effectivement bénéficier de ses effets de cache. De plus, certaines fonctionnalités opaques telles que les prefetchers peuvent améliorer – ou dégrader – les performances de la transformation sans que l'utilisateur puisse interagir avec ces dernières.

Nous pouvons ainsi conclure que les transformations de code pour les architectures de calcul généralistes actuelles posent des problématiques quant à la portabilité de méthodologies et l'efficacité énergétique.

Conclusion

Au cours de ce chapitre, nous avons abordés les défis logiciels lies à la programmation d'architectures de calcul, dits *non-von Neumann*. Nous avons abordé deux axes distincts : les interfaces pour la programmation d'applications, et la nature des applications numériques d'intérêt dans

l'art existant. Nous avons conclu que la standardisation de spécifications applicatives permettait de faciliter la portabilité d'applications standards vers des architectures de calcul émergentes, mais ne suffisait pas pour la programmation d'architectures arbitraires. Face à la limitation syntaxique de langages généralistes tels que le C ou encore le Fortran, les langages spécialisés (DSLs) permettent de spécifier des modèles d'exécution cohérents et adaptés aux architectures de calcul hétérogène.

Concernant la question de la détermination des applications numériques d'intérêt, nous avons attardé notre étude sur les codes de stencil, une classe d'algorithmes couramment employée dans une multitude de domaines. Nous avons, cependant, constaté que l'art antérieur proposait peu de solutions matérielles qui permettent l'accélération de codes de stencils généralistes. Les solutions existantes se focalisent sur l'accélération du calcul convolutif, ce qui ne représente qu'une portion des applications pouvant être implémentées par des codes de stencil.

Notre étude des solutions logicielles pour le calcul de stencil sur CPU et GPU présentait une multitude de DSLs et environnement de compilation pour optimiser l'ordonnancement des calculs ainsi que le format des données, et maximiser la consommation utile de leur bande passante. Cependant, nous avons pu conclure que ces transformations impliquent un surcoût non négligeable de l'empreinte mémoire des applications, qui en limite donc la portabilité selon les architectures ciblées, mais également l'efficacité énergétique par le phénomène de l'Energy Wall.

À la suite de cette étude bibliographique, nous pouvons émettre les observations suivantes :

1. **Les propositions architecturales dédiées à l'accélération de codes de stencil sont relativement limitées, comparés à la portée applicative de cette classe d'algorithmes.**
2. **Les architectures de calcul généralistes (CPU/GPU) conçus avec des systèmes mémoire pour le calcul haute-performance peuvent exécuter des codes de stencil généralistes fortement optimisés.**
3. **Ces optimisations, cependant, ne sont pas toujours réalisables pour des architectures basse-consommation et spécialisées pour du calcul frugal à faible empreinte énergétique.**

Ces deux observations seront nos aiguillages principaux pour l'élaboration des contributions proposés dans la présente thèse. Ces contributions se basent sur des travaux déjà existants, dont les problématiques abordées partagent des points communs aux observations susmentionnées. Le chapitre suivant présentera le cadre de ces travaux.

Chapitre

3 Background : l'architecture de calcul proche-mémoire et l'environnement de compilation dynamique

Introduction

Lors des précédents chapitres, nous avons présenté les problématiques liées à la conception d'architectures de calcul émergentes, et leur programmation afin d'exploiter leur potentiel de calcul de manière efficace tout en maintenant une productivité satisfaisante.

Nous présentons lors de ce chapitre deux travaux et en relation directe avec les contributions de la présente thèse.

Section 3.1 présente la SRAM Computationelle (C-SRAM), une architecture de calcul proche-mémoire qui permet d'effectuer des opérations arithmétiques et logiques vectorielles tout en minimisant le phénomène de bottleneck des architectures von Neumann.

Section 3.2 présente le paradigme de la compilation et de la spécialisation dynamique de code. Nous introduisons ensuite Hybrogen, un environnement de programmation dédié à la compilation dynamique de code.

Enfin, section 3.3 conclut ce chapitre en précisant les interactions de ces travaux avec les contributions de la présente thèse.

3.1 SRAM Computationelle (C-SRAM)

3.1.1 Principe général de la C-SRAM

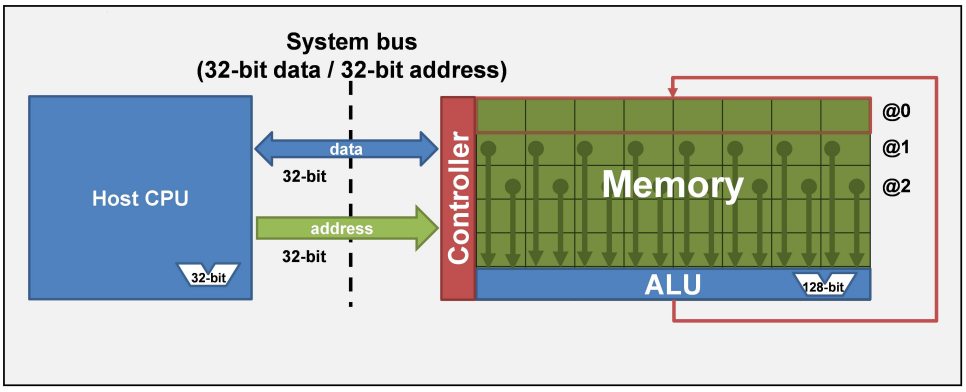


FIGURE 3.1 – Une architecture CPU intégrant une mémoire C-SRAM[KCT+18][GEK+20][DBH+21a][MCDK21][ENK+21][MCKD22].

La mémoire **SRAM Computationelle (C-SRAM)** (Figure 3.1) est une architecture mémoire développée au sein du **Laboratoire de Fonctions Innovantes pour circuits Mixtes** (LFIM) du CEA Grenoble[KCT+18][GEK+20][DBH+21a][MCDK21][ENK+21][MCKD22]. La C-SRAM est conçue à partir d'une mémoire volatile de type SRAM classique [FDB+10] à laquelle est ajouté un contrôleur mémoire sur-mesure ainsi qu'une ALU vectorielle. Ce concept de calcul proche-mémoire présente plusieurs intérêts. Premièrement, l'absence d'altérations aux cellules mémoires SRAM permet à la C-SRAM d'être implémentée sur la base de n'importe quelle architecture mémoire, facilitant alors son adoption au sein de solutions industrielles bénéficiant déjà de phases de tests et de validation extensives, tandis que la modification de cellules mémoires exige de reprendre toutes ces phases pour une nouvelle technologie. De plus l'intégration en périphérie d'une ALU vectorielle est suffisamment proche de la mémoire pour permettre de minimiser le phénomène de bottleneck au sein des architectures von Neumann, grâce à la substitution d'échanges de données entre le CPU et la C-SRAM par l'envoi d'instructions vectorielles.

3.1.2 Jeu d'instructions de la C-SRAM

Operation type	Instruction format	operations
Memory	R	copyeq, copyneq, copygeq, copylt, copyleq, copygt
	I	copy
	U	bcast
Logical	R	and, or, xor, nand, nor, xnor
	I	not, sll, srl
Arithmetic	R	add, sub, mul, mac, cmp

TABLE 3.1 – Liste des instructions C-SRAM, classifiées par type d'opérations puis par format.

La table 3.1 présente le jeu d'instructions de la mémoire C-SRAM. Ce jeu d'instructions est initialement divisé en trois catégories : les instructions logiques, les instructions arithmétiques et les instructions mémoires. Pour toutes ces instructions, les opérations sont effectuées entre rangées au sein de la mémoire C-SRAM.

Les instructions logiques effectuent des opérations booléennes entre deux rangées de la mémoire C-SRAM et en stockent les résultats dans une troisième rangée.

Les instructions arithmétiques effectuent des opérations arithmétiques vectorielles, dont les tailles de données peuvent être sélectionnées selon la déclinaison d'instruction employée. Par exemple, l'instruction d'addition vectorielle `add` possède trois déclinaisons : `add8`, `add16` et `add32`. Chacune de ces déclinaisons effectue une addition de vecteurs dont les éléments ont des tailles de données différentes : respectivement 8, 16 et 32 bits.

La sélection des tailles des éléments pour les opérations arithmétiques vectorielles a un impact direct sur la ***cardinalité*** de ces opérations. Nous définissons la cardinalité des opérations arithmétiques vectorielles comme le nombre d'éléments au sein du vecteur. La cardinalité d'un vecteur est à distinguer de sa taille, qui est dépendante de son implémentation architecturale.

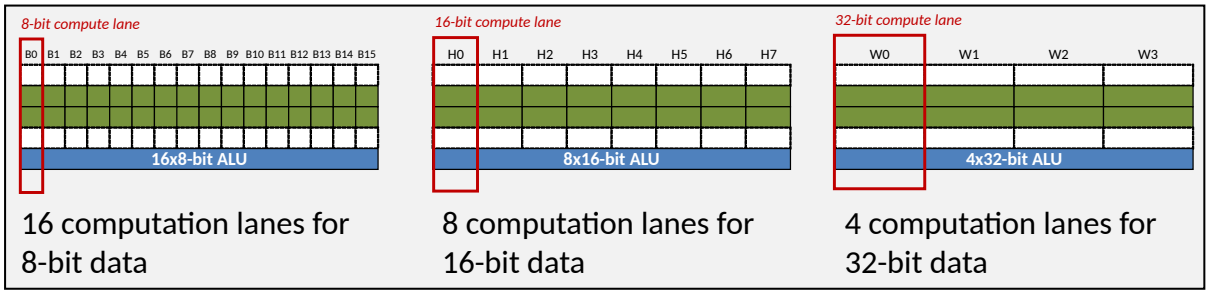


FIGURE 3.2 – Degré du parallélisme des calculs de la C-SRAM en fonction du paramétrage de l'ALU interne.

La Figure 3.2 présente l'exemple d'une C-SRAM dont les tailles de vecteurs sont de 128 bits. Selon l'utilisation d'instructions 8-bit, 16-bit ou 32-bit, la mémoire C-SRAM peut effectuer des opérations arithmétiques vectorielles de cardinalité 16, 8, ou 4.

Enfin, Les instructions mémoires permettent de gérer de façon interne les données stockées au sein de la mémoire C-SRAM, ou d'initialiser des rangées à des valeurs spécifiques.

Il y a trois sous-catégories d'instructions mémoires. L'instruction de copie standard permet de recopier le contenu d'une rangée source vers une rangée destination. Les instructions de copie conditionnelle sont commandées à l'aide de l'instruction de comparaison arithmétique vectorielle `cmp`, et permettent de recopier le contenu d'une rangée source vers une rangée destination selon si les conditions attendues par l'instruction appelée sont vérifiées par le résultat de l'instruction `cmp`. Enfin, l'instruction de *broadcast* (`bcast`) permet d'initialiser tous les éléments au sein d'une rangée destination à une valeur passée en paramètre par le HPU. Les instructions de broadcast et de copie conditionnelle sont déclinées selon plusieurs tailles de données, comme les instructions arithmétiques vectorielles.

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Format R	1	0	0	0	0	0	Op. code								Destination																0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Source op. 2																Source op. 1															

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Format I	1	0	0	0	0	0	Op. code								Destination																0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	16-bit immediate																Source op. 1															

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Format U	1	0	0	0	0	0	Op. code								Destination																0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	32-bit immediate																															

TABLE 3.2 – Les différents formats d'encodage des instructions C-SRAM 64-bit.

La Table 3.2 présente les formats d'encodage du jeu d'instructions de la C-SRAM. Les instructions de format R prennent en paramètres en opérandes deux rangées source. Les instructions de format I prennent en opérandes une rangée source un immédiat encodé sur 16 bits. Enfin, les instructions de format U prennent un immédiat sur 32 bits.

Le jeu d'instructions de la C-SRAM est encodé sur 64 bits. Chaque champ correspondant a l'indexation de rangées, source ou destination, possède 16 bits.

3.1.3 Support des instructions C-SRAM par une architecture hôte

La mémoire C-SRAM peut être commandée par le HPU selon deux modes de fonctionnement : un mode *mémoire* et un mode *calcul*. L'activation de ces deux modes dépend de la valeur du bit de poids fort du bus d'adresse lorsque le HPU envoie une requête mémoire à la C-SRAM, avec la valeur '1' correspondant au mode calcul. Cela signifie que les instructions C-SRAM sont envoyées par le HPU sous la forme de requêtes mémoires à sa destination.

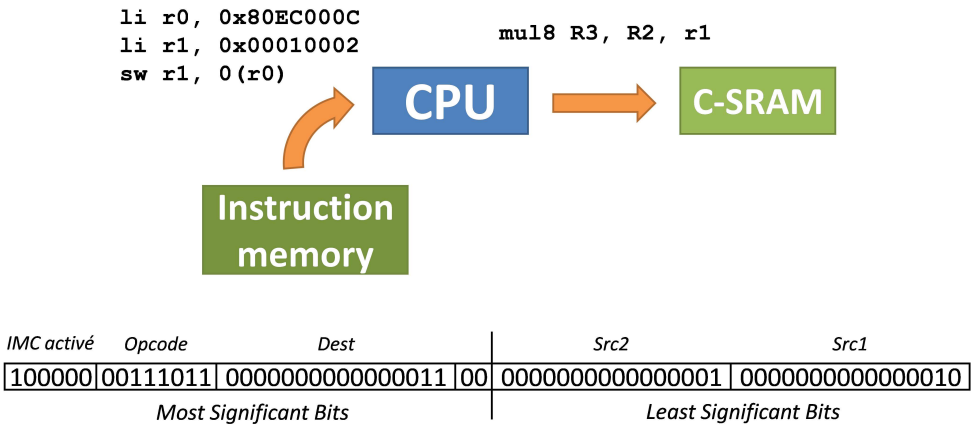


FIGURE 3.3 – Le HPU envoie une instruction de multiplication vectorielle 8-bit à la mémoire C-SRAM. La préparation de cette requête nécessite plusieurs instructions HPU.

La Figure 3.3 montre un exemple de programmation de la C-SRAM par le HPU. L'instruction C-SRAM souhaité est la multiplication vectorielle 8-bit des rangées R1 et R2, a destination de la rangée R3. Pour envoyer cette requête d'opération à la C-SRAM, le HPU va exécuter une instruction d'écriture mémoire *Store-Word* (`sw`) dont l'adresse correspond aux bits de poids forts de l'instruction C-SRAM et la valeur envoyée aux bits de poids faibles. Ainsi, le HPU peut envoyer des instructions C-SRAM sur 64-bit, conformément aux spécifications de l'encodage du jeu d'instructions C-SRAM.

L'utilisation du bit de poids fort du bus d'adresse implique qu'une partie de l'espace logique du HPU est réservée pour pouvoir correspondre à la plage de programmation du jeu d'instructions de la C-SRAM.

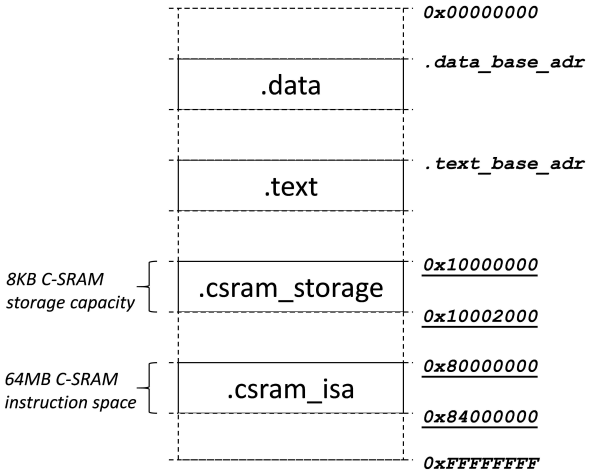


FIGURE 3.4 – Espace d'adressage logique du CPU hôte suite à l'intégration de la C-SRAM.

La Figure 3.4 présente le contenu de l'espace d'adressage logique d'un HPU possédant une mémoire C-SRAM de taille 8 Kilo-octets. Les segments `.data` et `.text` correspondent respectivement aux données et aux instructions du programme exécuté par le HPU. Le segment `.csram_storage` correspond au stockage interne – c'est-à-dire la grille de cellules SRAM – de la mémoire C-SRAM. La taille de ce segment est dependant de la capacité de stockage de la C-SRAM, ici de 8 Kilo-octets. Enfin, le segment `.csram_isa` correspond à la plage de programmation du jeu d'instructions de la C-SRAM. Cette plage est de taille constante, de par les spécifications de la C-SRAM, qui est estimée à 16 Kilo-octets au minimum.

L'interface définie par les spécifications de la C-SRAM lui permet d'être intégrée au sein de n'importe quelle architecture hôte, de façon similaire a des périphériques programmés par des registres de contrôle. Contrairement aux périphériques, cependant, la mémoire C-SRAM utilise une partie de l'espace d'adressage logique de l'architecture hôte pour améliorer la latence de programmation des opérations.

3.1.4 Propositions scientifiques basées la C-SRAM

Les spécifications de la C-SRAM présentées lors des précédentes sous-sections décrivent le fonctionnement de l'unité fondamentale de mémoire C-SRAM que nous appelons une *tuile*.

Sur la base de ces spécifications, des propositions architecturales ont été proposées et publiées pour intégrer cette tuile dans des systèmes complets. Dans la suite de cette sous-section, nous présentons deux intégrations de la C-SRAM.

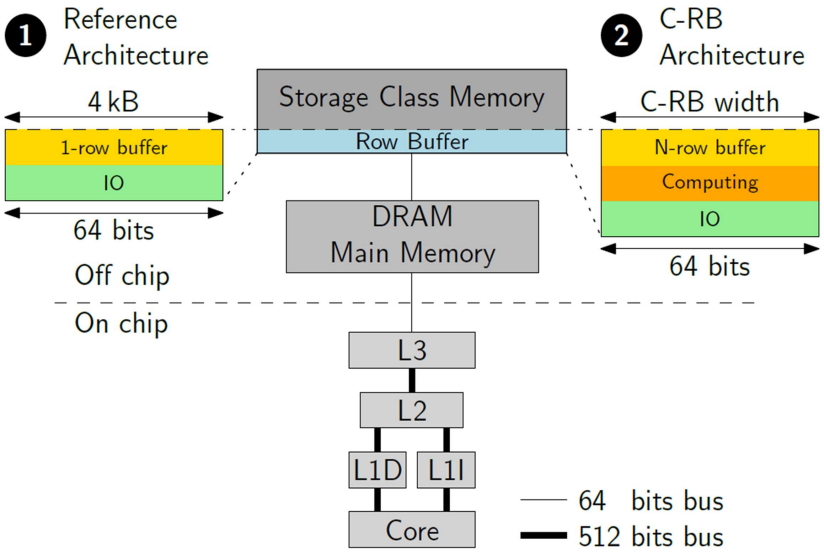


FIGURE 3.5 – Intégration de mémoire C-SRAM en tant que *Computing Row Buffer* (C-RB) pour les *Storage-Class Memories*, image tirée de [ENK+21].

Le ***Computing Row-Buffer*** (C-RB)[ENK+21] est l'intégration de la mémoire C-SRAM en périphérie de *Storage-Class Memories* (SCMs), des mémoires de stockage intermédiaires à la mémoire principale et aux unités de stockage froid tels que les disques durs. Les SCMs utilisent

des technologies mémoires dont l'endurance aux lectures et écritures successives est limitée, et pour lesquelles il est donc nécessaires d'intégrer des *Row Buffers*, des tampons mémoire conçus dans des technologies plus endurantes pour *cacher* les données lues dans des plages à la granularité de rangées.

La motivation derrière la proposition du C-RB est l'aggravation du coût énergétique du transfert des données de la SCM vers le CPU, au sein d'une hiérarchie mémoire traditionnelle. L'intégration du calcul proche-SCM permet donc d'effectuer au plus proche du stockage les opérations arithmétiques et logiques supportées par l'architecture C-SRAM tout en mitigant le phénomène d'*Energy Wall* (Figure 3.5).

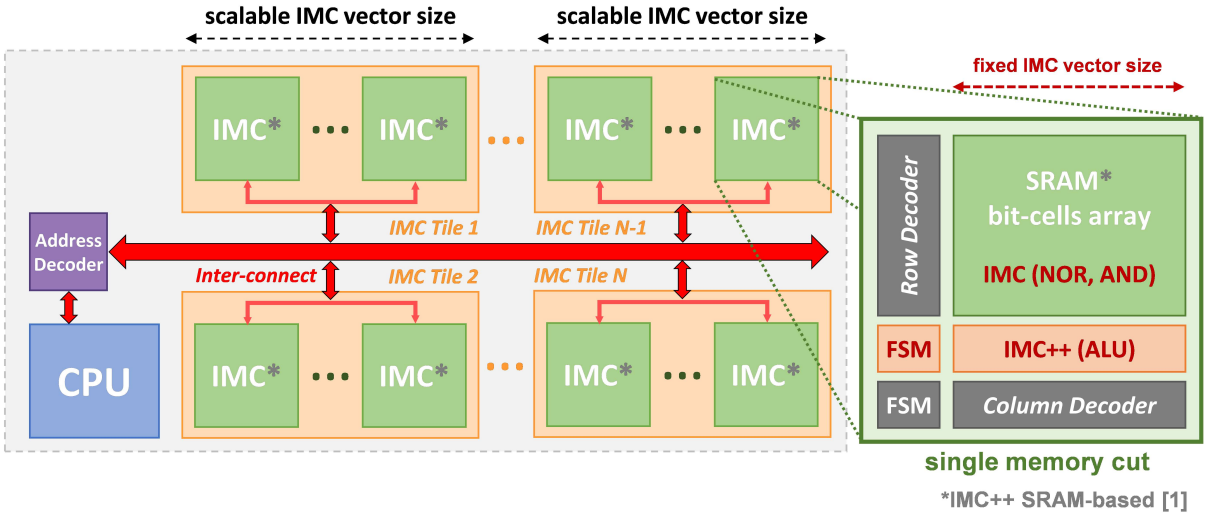


FIGURE 3.6 – L'architecture METEOR permet la mise à l'échelle de l'intégration massive de la C-SRAM tout en offrant un degré de parallelisme reconfigurable à la volée, image extraite de [GEK+20].

Une tuile C-SRAM peut être conçue et implémentée pour posséder une certaine capacité de stockage ainsi qu'un certain degré de parallélisme de calcul. Cependant, ces paramètres sont limités par des contraintes physiques qui peuvent en impacter la latence ainsi que la consommation énergétique. L'architecture **Matrix of Elementary Tiles Enabling Optimal Reconfigurability** (METEOR)[GEK+20] propose l'interconnexion de plusieurs tuiles afin de permettre l'intégration massive de mémoire C-SRAM. Cet interconnect peut également être paramétré pour programmer l'exécution d'instructions C-SRAM sur plusieurs tuiles de façon reconfigurable, ce qui permet d'adapter le degré de parallélisme global de METEOR selon les besoins logiciels (Figure 3.6).

Notons que ces propositions architecturales n'altèrent pas de façons significatives le modèle d'interface de la C-SRAM. Les travaux de la présente thèse traiteront donc les problématiques de la programmation de l'architecture C-SRAM et, par extension, des propositions architecturales susmentionnées.

3.2 Génération et spécialisation dynamique de code

3.2.1 Spécialisation de code

Le cycle de vie classique d'une application est borné par son implémentation sur la base d'un langage de programmation jusqu'à l'exécution de son binaire sur la plateforme visée. Au centre de ce cycle de vie se trouvent les compilateurs qui ont la charge de transformer l'implémentation de l'application humainement lisible en code binaire adapté pour le langage machine. Bien que ces compilateurs étaient initialement conçus pour générer du code machine et l'assembler au sein du binaire, ces programmes ont été développés également pour effectuer des optimisations automatiques, à la suite d'analyse statique de code. De nombreux travaux de la littérature ont proposé des optimisations logicielles afin d'améliorer des applications selon des critères donnés. Ces critères peuvent être variés, visant la minimisation du temps d'exécution du code ou encore la minimisation de la taille du code.

Optimisation	Exemple

Réduction de force	`// Often slow implementation, due to the overhead of the MUL`
	`a = b * 8;`
	`// faster implementation`
	`a = b >> 3;`

Constant folding	`// The following statement:`
	`i = 3 * 4;`
	`// is pre-computed at compile-time as:`
	`i = 12;`

Déroulage de boucle	`// Example loop, original`
	`for (int i = 0; i < N; i += 1)`
	`{`
	`  y[i] = a[i]*alpha + y[i];`
	`}`
	`// Example loop, twice unrolled`
	`for (int i = 0; i < N; i += 2)`
	`{`
	`  y[i  ] = a[i  ]*alpha + y[i  ];`
	`  y[i+1] = a[i+1]*alpha + y[i+1];`
	`}`

TABLE 3.3 – Quelques optimisations de code résolubles lors de l'analyse statique d'un code d'entrée.

La Table 3.3 présente une liste non exhaustive d'optimisations pouvant être appliqué sur un code d'entrée par un compilateur logiciel. Ces optimisations sont appliquées selon la résolution de problèmes pouvant être énoncés ainsi :

— "Est-il possible d'améliorer l'*efficacité* d'exécution du code ?" Face à cette problématique, un compilateur va tenter de remplacer une instruction ou une séquence d'instructions

par un ensemble d'instructions équivalentes au niveau fonctionnel mais qui présente de meilleures performances d'exécution sur l'architecture de calcul ciblée.

Nous pouvons citer le *Constant folding*, qui vise à identifier des clauses de calcul pouvant être précalculées lors de la phase de compilation et assignée au registre cible pour un résultat équivalent.

La réduction de force est une autre optimisation de code couramment utilisée, qui transforme une opération de calcul donnée par une séquence d'opérations présentant de meilleures performances lors de l'exécution sur l'architecture ciblée. Une réduction de force courante est la transformation de multiplication ou de division par des nombres puissances de 2, en instructions de décalage arithmétique respectivement vers la gauche et vers la droite. La raison est que très souvent, les opérateurs matériels de multiplication et de division possède une latence d'opération plus élevée que les autres opérateurs à disposition de l'ALU.

Nous mentionnons ces optimisations avec pour intérêt principal l'amélioration de la latence d'exécution globale des applications, mais il est envisageable pour les développeurs de de décrire d'autres critères qualitatifs concernant l'*efficacité* du code, telles que la consommation énergétique.

— "Est-il possible d'améliorer la taille du code ?" Cette problématique peut souvent être résolue lors de la construction de l'objet binaire de l'application avec les *Link-Time Optimisations* (LTO). Ces optimisations vont réduire au maximum la taille du code en étudiant pour chaque section de code machine diverses caractéristiques. Par exemple, une section de code machine redondante avec une autre pourra être supprimée, de même qu'une section de code dite *morte*, qui n'est jamais accédée lors de l'exécution. Ces types de code sont très courants de par le paradigme de programmation partitionnée par des librairies.

Notons également que certaines optimisations de code visant à améliorer l'efficacité d'exécution du code peuvent entrer directement en conflit avec la problématique d'en améliorer la taille. Ainsi, le déroulage de boucles est une optimisation qui vise à réduire le nombre d'itérations d'une boucle donnée en en dupliquant les clauses internes. Cette optimisation est très intéressante car très souvent, les instructions de branchement sont particulièrement coûteuses en latence. Ainsi, dans l'exemple présent à la Table 3.3, la boucle est déroulée deux fois. On dit alors qu'elle subit un déroulage de boucle de degré 2. Cependant, bien qu'un déroulage de boucles de degré maximal – c'est-à-dire égal à la borne maximale de la boucle – est envisageable, cette optimisation augmente la taille effective de la boucle et donc la taille de l'application. Le déroulage de boucles est donc une optimisation de code qui, lorsqu'elle est envisageable, doit être paramétrée avec une certaine finesse pour compromettre l'augmentation de la taille du code pour l'amélioration de la performance.

3.2.2 Compilation dynamique de code

Il existe donc de nombreuses techniques d'optimisation de code qui permettent d'en améliorer l'efficacité selon différents critères. Cependant, ces optimisations doivent être appliquées dans la majorité avec le respect de spécifications fournies par le langage de programmation qui interface l'utilisateur avec le matériel.

```c
int func(int a, int b, int alpha)
{
  return a + b * alpha;
}

int main(void)
{
  int a = 3;
  int b = 5;
  int alpha = 6;
  res = func(a, b, alpha);
  return 0;
}
```

(a) Le constant folding ainsi que l'inlining sont réalisables.

```c
int func(int a, int b, int alpha)
{
  return a + b * alpha;
}

int main(void)
{
  int a = 3;
  int b = 5;
  volatile int alpha;
  res = func(a, b, alpha);
  return 0;
}
```

(b) La non-résolution de la variable `alpha` ne permet pas le constant folding ici.

FIGURE 3.7 – Les variables aux valeurs insolubles lors de la phase d'optimisation statique du code limite le spectre des optimisations applicables.

Un exemple de spécification de code qui limite la portée des optimisations est, par exemple, les variables préfixées du mot-clé `volatile`. Les variables dites *volatile* ne promettent aucune garantie concernant l'état de leur valeur en mémoire, et peuvent être modifiées de façon autonome du code écrit par le programmeur, par le biais d'effets de bord. En conséquence, le compilateur ne peut pas prendre de suppositions sur l'utilisation de la variable et va limiter les optimisations réalisables sur cette dernière.

Dans l'exemple présenté à la Figure 3.7, le code de droite ne peut pas être optimisé avec des techniques telles que le *constant folding*, à cause de l'attribut *volatile*.

Les effets de bord sont à rattacher au concept de *fonctions pures*, des fonctions qui ne possèdent aucun effet de bord que le retour de leur valeur résultat, et qui n'appellent elles-mêmes aucune fonction *impure*[Fou]. S'il est possible d'assister le compilateur dans la détection de fonctions pures, avec des attributs de fonctions tels que `__attribute__((pure))` pour le compilateur GCC, l'utilisation de ressources telles que les périphériques ou les segments de mémoire partagée induisent *de facto* l'impureté des fonctions, et donc la limitation du jeu d'optimisation applicable sur ces derniers.

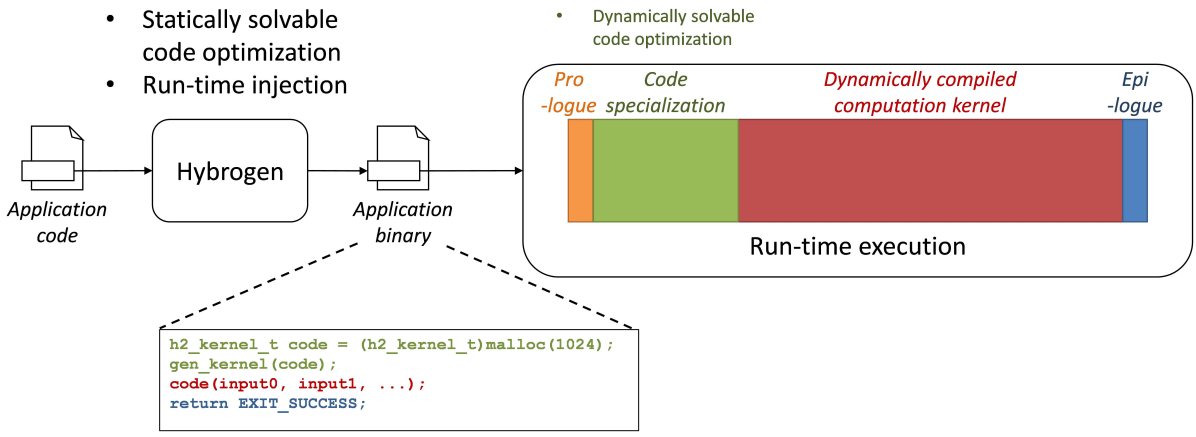
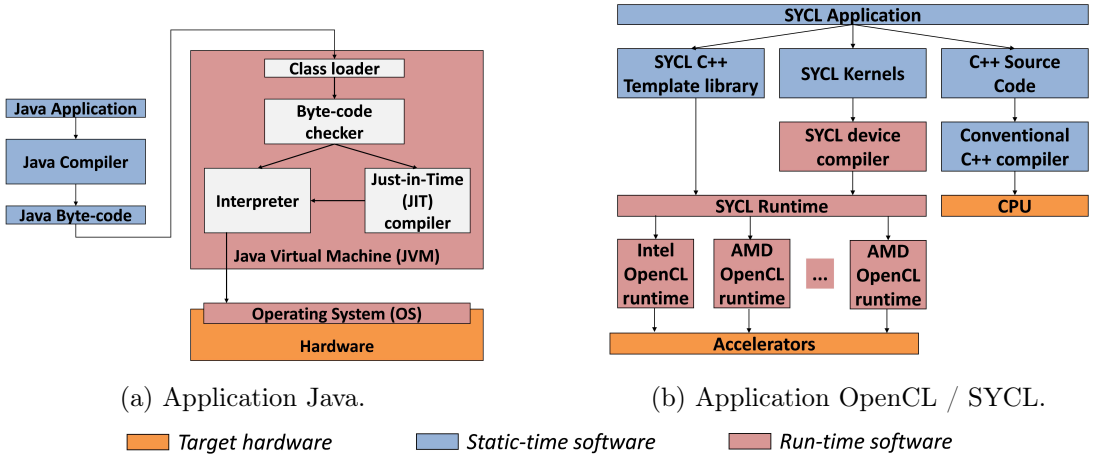


FIGURE 3.8 – La compilation dynamique de code par le biais de Hybrogen permet de réaliser des optimisations de code autrement insolubles lors de la compilation statique[DCMK21].

La **compilation dynamique** vise à améliorer la performance des applications par la génération du code lors de l'exécution du programme. Ce paradigme d'optimisation permet l'analyse de code a des temps d'exécution où les statuts de variables, autrement insolubles lors d'une phase de compilation statique, peuvent être résolus (Figure 3.8). Le mécanisme de génération et d'optimisation de code idéal doit pouvoir implémenter un jeu d'optimisations qui améliore l'exécution du code ciblé avec le moins de surcoûts d'exécution. Des optimisations éligibles à de tels critères incluent l'élimination de code redondant, la réduction de force ou encore le constant folding précédemment présenté.

De nombreux travaux ont été mis en place pour étudier le support et les bénéfices de la compilation dynamique[APC+96][KGSC01][KAH07]. Plusieurs travaux ont également étudié des solutions de support de la compilation dynamique auprès des programmeurs par le biais de langages spécialisés[MCE00][GMP+00][CCL+14].

La compilation dynamique de code est une stratégie employée au sein de paradigmes de compilations tels que la compilation à la volée *Just-in-Time*, avec pour principal représentant Java. L'utilisation de bytecode architecture-indépendant permet à une application Java d'être déployée sur n'importe quelle architecture de calcul disposant d'un environnement d'exécution Java.



(a) Application Java. (b) Application OpenCL / SYCL.

FIGURE 3.9 – Comparaison du cycle de vie d'une application Java (orientée CPU) et d'une application OpenCL (orientée accélérateurs).

Le paradigme de la compilation dynamique de code est une alternative intéressante à la compilation statique de code pour la programmation de nœuds de calcul hétérogènes. En effet, la variété des opérations arithmétiques de ces architectures offre de nombreuses opportunités d'optimisation, qui peuvent cependant être paramétrées ou calibrés selon les applications ainsi que les entrées de ces dernières. Il existe bien des modèles de programmation à destination d'architectures hétérogènes, tels que OpenCL et SyCL, qui emploient par souci de portabilité la génération dynamique de code. Cependant, ces environnements d'exécution sont spécialisés pour la gestion des nœuds de calcul hétérogènes et ne permettent pas d'optimiser dynamiquement le code à destination des accélérateurs à d'autres étapes de temps que l'initialisation de l'environnement. Il serait cependant difficile de mettre en place un flot de simulation conjoignant l'ensemble des fonctionnalités d'environnements tels que Java et SYCL sans induire un surcoût d'exécution non-négligeable(Figure 3.9).

C'est à la suite de ces motivations que *Hybrogen*, un environnement de compilation dynamique pour architectures hétérogènes, a été mis au point.

3.2.3 Environnement de compilation dynamique de code Hybrogen

Hybrogen[DCMK21] est un environnement de programmation dédié à la compilation dynamique de code à destination des architectures hétérogènes. Ce projet hérite des travaux introduits par la recherche et le développement de `deGoal`[CCL+14], un environnement de programmation dédié à la compilation dynamique de code pour architectures CPU. Pour permettre l'intégration continue d'architectures hétérogènes émergentes au sein de l'environnement, Hybrogen intègre un système de bases de données pour la gestion d'architectures (Figure 3.10). Cette base de données est alimentée par le biais de descriptions architecturales, qui décrivent sous un format cohésif et humainement lisible les caractéristiques de l'architecture. Ces caractéristiques incluent le jeu d'instructions, ou encore les registres disponibles auprès du logiciel.

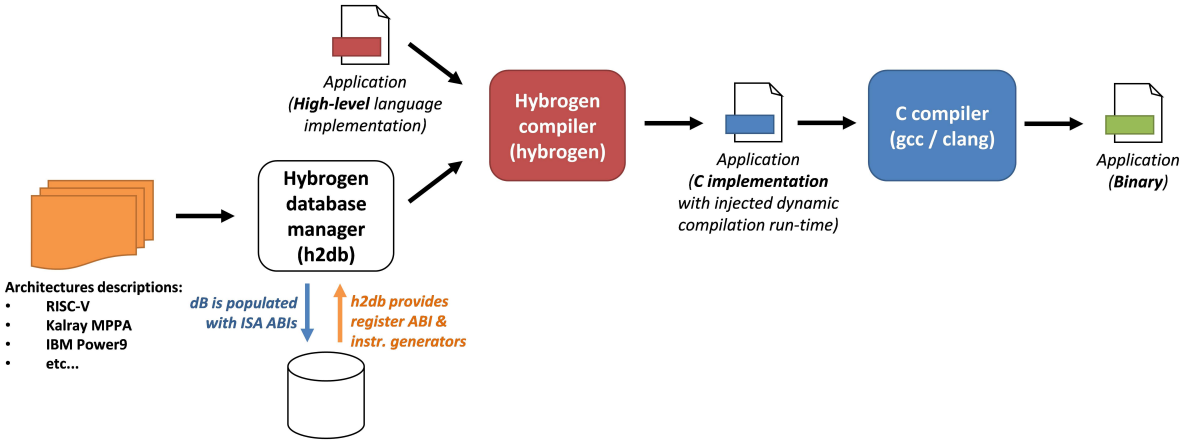


FIGURE 3.10 – Le flot de compilation de Hybrogen permet de programmer des kernels dynamiquement compilés.

Pour la description de codes dynamiquement compilés, l'environnement Hybrogen fournit un langage spécialisé : le `hybrolang`. La syntaxe de ce langage est inspirée du langage C, de par son paradigme de programmation impératif ainsi que les types primitifs supportés, tels que `void`, `int` et `float`. HybroLang ajoute, cependant, la possibilité de paramétrer ces types primitifs pour représenter des données vectorielles, et ainsi offrir aux programmeurs un support explicites pour des architectures de calcul SIMD. Par exemple, un vecteur contenant seize éléments de 8 bits est défini par : `int 8 16` s'il est stocké en registre, et `int[] 8 16` s'il est stocké en mémoire. HybroLang supporte également des types de données spéciaux tels que les entiers saturés (de type primitifs `sint`) pour ajouter le support explicite des opérations arithmétiques saturées. Enfin, HybroLang permet de définir des *constantes dynamiques*, des constantes dont la valeur n'est définie que lors du runtime et qui peut ensuite être intégrée de façon statique au code dynamiquement compilé.

La Figure 3.11 présente un exemple pratique de code utilisant du HybroLang pour le rendre dynamiquement compilé. Le code en question est une implémentation de la routine `axpy`, une routine faisant partie de la spécification du BLAS et qui effectue le calcul $y \leftarrow \alpha \times x + y$, avec α un scalaire et [x,y] des vecteurs arithmétiques. L'implémentation de la routine se décompose en deux parties : le générateur de code dynamiquement compilé, écrit en C et nommé `gen_axpy`, et la définition de la fonction à compiler dynamiquement, écrite en HybroLang et nommée `axpy`. La section de code écrite en HybroLang correspond à la section située entre les deux balises de définition de HybroLang, `#[` et `]#`.

Le générateur de code `gen_axpy` est défini comme un pointeur vers une fonction de type `(gen_axpy_t)(int, int *, int *)`, ce qui veut dire que la fonction dynamiquement compilée devra prendre en paramètres un entier scalaire et deux pointeurs vers des entiers scalaires. Dans le cas du présent exemple, `gen_axpy` prend trois paramètres en entrées : un pointeur vers la zone mémoire dans laquelle générer le code, et deux entiers scalaires, qui sont utilisé au sein d'`axpy` comme constantes dynamiques. La première constante dynamique est `size`, qui correspond à la taille des vecteurs arithmétiques passés en paramètres. La seconde

```
typedef (*gen_axpy_t)(int, int *, int *);
gen_axpy_t gen_axpy(h2_insn_t *ptr, int size, int cardinal)
{
#[
  void axpy(int 32 1 A, int[] 32 1 X, int[] 32 1 Y)
  {
    int 32 1 i;
    int 32 #(vec_len) a, x, y;
    a = A;
    for (i = 0; i < #(size); i += #(cardinal))
    {
      x = X[i*#(cardinal)];
      y = Y[i*#(cardinal)];
      Y[i*#(cardinal)] = a * x + y;
    }
  }
]#
  return (gen_axpy_t)ptr;
}
```

FIGURE 3.11 – Implémentation de la routine `axpy` en langage `hybrolang`. Les vecteurs d'entrée en de sortie sont paramétrés de façon dynamique.

constante dynamique est `cardinal`, et elle correspond à la cardinalité à laquelle le programmeur souhaite vectoriser le code.

Comme mentionné précédemment, la fonction `axpy` prend trois paramètres : un scalaire A et deux pointeurs vers des tableaux de scalaires X et Y. Cette dernière déclare trois variables vectorielles `a`, `x` et `y`, dont la cardinalité est définie par `cardinal` (les constantes dynamiques étant invoquées entre parenthèses et précédées d'un #). La boucle qui itère sur X et Y est bornée entre 0 et `size`, et incrémentée de `cardinal` éléments. Lors de la compilation dynamique de code, les instructions associées aux opérations arithmétiques sur les vecteurs `[a,x,y]` seront sélectionnées en fonction de `cardinal`, une fois sa valeur déterminée au runtime.

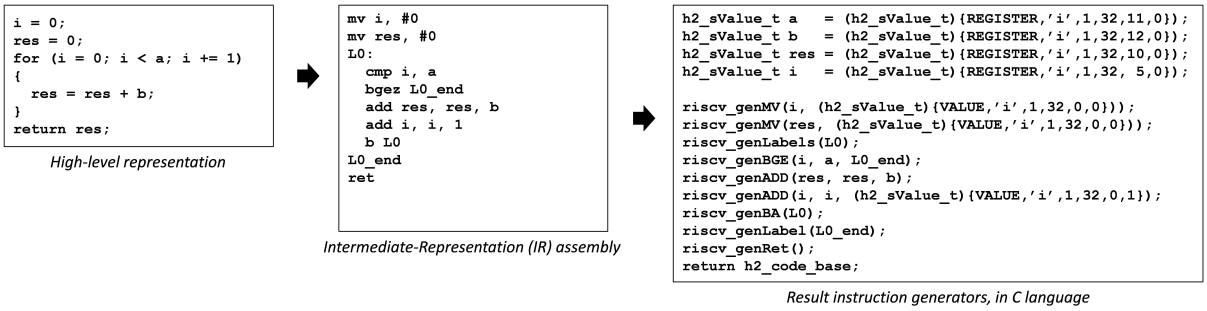


FIGURE 3.12 – Flot de génération des générateurs d'instructions exécutés au run-time lors de la phase de compilation dynamique.

La Figure 3.12 présente les trois phases de compilation d'une application intégrant du code `hybrolang`. Une première phase de compilation statique transforme la Représentation Intermé-

diaire issue du HybroLang en générateurs d'instructions, écrits dans la syntaxe du langage hôte. Ces générateurs permettent la spécialisation conditionnelle des instructions en fonction des données dynamiques de code. Une deuxième phase de compilation est alors nécessaire pour générer le code machine de l'application. Cette méthodologie d'intégration de la compilation dynamique peut être employée sur la base de n'importe quel langage hôte, tel que le Python ou encore le javascript, pour permettre la programmation explicite de la génération et la spécialisation dynamique de code de façon transparente, vis-à-vis des développeurs.

3.3 Discussion et observations

Nous avons présenté la SRAM Computationelle (C-SRAM), une architecture de calcul proche-mémoire aux capacités arithmétiques vectorielles. Le modèle d'interface de la C-SRAM utilise le bus d'adresse et de données du bus système CPU pour permettre la programmation efficace d'instructions de calcul proche-mémoire sans modifier de façon substantielle l'architecture hôte.

Nous avons également introduit le paradigme de la compilation dynamique, qui permet l'adaptation et l'optimisation du code lors de l'exécution de l'application. Nous avons présenté Hybrogen, un environnement de programmation dédié à la compilation dynamique. Les fonctionnalités de Hybrogen, tels que la gestion de multiples architectures de calcul hétérogènes, ainsi que le langage spécialisé `hybrolang`, ont été introduits.

Ces deux projets de recherche utilisent des axes d'approche distincts (logiciels et matériels) pour résoudre les problèmes de programmation d'architectures hétérogènes et de contention de la mémoire. Les contributions de la présente thèse se baseront sur ces projets pour mettre au point des solutions qui permettent de programmer efficacement une architecture de calcul proche-mémoire – et dans notre cas précis, la C-SRAM. Elles aborderont les questions scientifiques suivantes :

- Comment efficacement programmer la mémoire C-SRAM pour maximiser la performance d'applications ?
- Comment permettre à la mémoire C-SRAM de bénéficier de la compilation dynamique de code pour maximiser la performance de ses applications lors de l'exécution ?

Chapitre

4 Dispositif de transfert intelligent à destination des architectures de calcul proche-mémoire

- *Quels sont les composants nécessaires pour l'accélération de codes de stencil ?*
- *Quelles sont les interfaces nécessaires pour intégrer ces composants à un système donné ?*

Introduction

Nous présentons dans le présent chapitre le ***Data-locality Management Unit (DMU)*** une solution matérielle qui permettent de transférer efficacement des données de stencils, d'une mémoire principale vers une mémoire IMC généraliste. Cette solution dispose d'un jeu d'instructions permettant la programmation de transferts pour n'importe quel voisinage de stencil arbitraire, et assure la minimisation de la consommation de la bande passante de la mémoire principale. Cette solution est la seule de l'état de l'art, à notre connaissance, à proposer un bloc de transferts dédié à cette tâche qui puisse être intégrée dans plusieurs architectures de calcul généralistes – tout particulièrement les architectures disposant d'une unité de calcul vectoriel et d'une mémoire programmable, offrant donc aux architectes matérielles une plus large flexibilité d'adoption.

Le chapitre est organisé comme suit : Section 4.1 présente de façon générale le DMU. Section 4.2 présente le jeu d'instructions. Section 4.3 présente la fonctionnalité du DMU par le biais d'exemples. Enfin, Section 4.4 fera la conclusion de chapitre et relèvera des observations sur la solution proposée.

4.1 Présentation du DMU

Le Data-locality Management Unit (DMU) est un bloc de transferts de données qui permet de transférer des données au sein d'un système complet, entre une mémoire DRAM et une mémoire C-SRAM. Pour être programmé, le DMU intègre un jeu d'instructions qui permettent au HPU de programmer des transferts de données. Ces instructions peuvent être paramétrées pour transférer et réorganiser des données de stencil de la mémoire DRAM vers la mémoire C-SRAM. Cette réorganisation de données est effectuée de telle sorte à faciliter le calcul vectoriel de codes de stencils au sein de la mémoire C-SRAM. Enfin, le DMU dispose d'un bus d'échange de données dédié entre ce dernier, la mémoire C-SRAM et la mémoire DRAM, ce qui permet au DMU de transférer des données de façon non-bloquante.

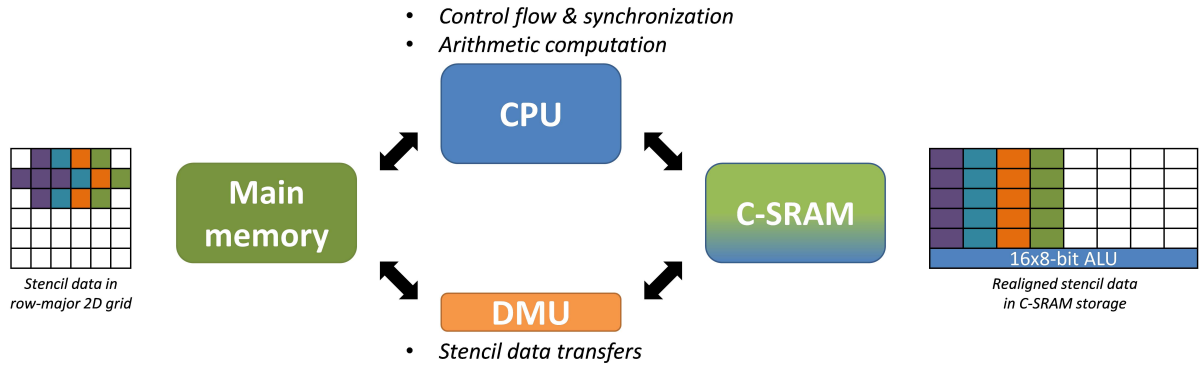


FIGURE 4.1 – L'intégration du Data-locality Management Unit (DMU) permet de décharger le HPU de la tâche de transférer et réorganiser les données de stencils tout en le permettant de paralléliser le transfert de données avec du calcul en C-SRAM.

La Figure 4.1 présente un exemple de transfert de données avec le DMU. Dans cet exemple, les données sont transférées depuis une structure de données à deux dimensions, stockée en mémoire DRAM, vers la mémoire C-SRAM. Le DMU est programmé pour transférer quatre voisinages de stencil, dont la forme est une croix à cinq points (aussi appelé stencil *de von Neumann*[TM87]). Chacune de ces croix de stencil est réorganisée lors de la réécriture en mémoire C-SRAM pour que ses données soient transposées verticalement. Cette transposition verticale permet ainsi le calcul vectoriel de codes de stencil au sein de la mémoire C-SRAM.

4.2 Jeu d'instructions du DMU

À partir de la définition générale d'un algorithme de stencil, nous établissons les paramètres nécessaires à la programmation d'un transfert de données de stencil.

Nous rappelons ainsi qu'un code de stencil est un algorithme qui met à jour les *différents points* d'une **grille d'entrée** vers une **grille de sortie**, selon un **voisinage à la géométrie définie**.

Les données d'intérêt du code de stencil seront lues à partir d'un **point de départ** à un pas de lecture constant pour toute la durée du transfert. Ce pas de lecture est couramment appelé **stride de lecture** dans le contexte des accès mémoire, et peut être corrélé aux *strides* d'application de filtres dans le contexte de la vision par ordinateur et du traitement d'image. Le transfert doit également être paramétré avec la taille des éléments au sein de la structure de données ciblée, parmi 8, 16 ou 32 bits. Enfin, la **géométrie** des voisinages de stencil à transférer ainsi que leur nombre doit être paramétré.

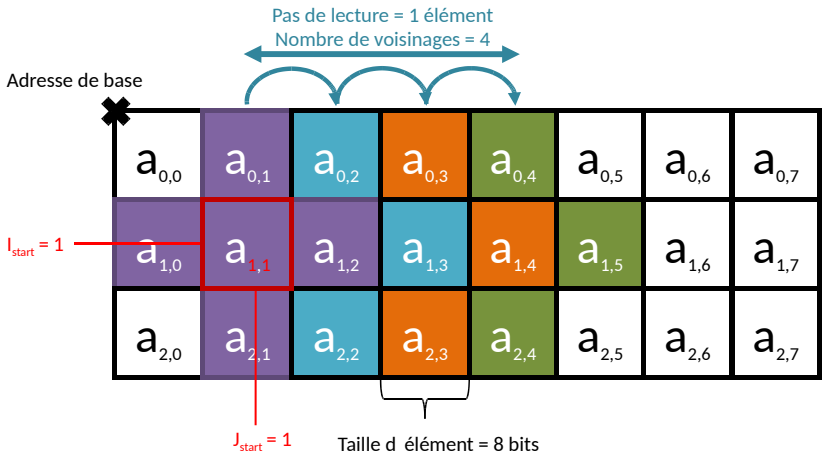


FIGURE 4.2 – Première partie des paramètres du transfert de données présenté lors de la Figure 4.1. Les croix de stencil d'éléments 8 bits sont lues depuis la mémoire DRAM avec un pas de lecture d'un élément.

La Figure 4.2 reprend l'exemple du transfert de données présenté lors de la Figure 4.1, en explicitant les paramètres présentés lors du précédent paragraphe. La géométrie des voisinages prend la forme de croix à cinq points. Chaque point correspond à une donnée d'un octet en taille, et les voisinages sont lus à partir du point de coordonnées $(I, J) = (1, 1)$. Au terme du transfert, quatre croix de stencils seront transférées dans le stockage de la C-SRAM. Ces croix sont lues à un pas de lecture égal à 1, ce qui implique par leur géométrie des superpositions des voisinages successifs, et une redondance de données. Nous expliquerons plus tard notre procédé pour éviter les lectures redondantes depuis la mémoire DRAM et optimiser de façon globale le transfert des données.

Nous ajoutons parmi la liste des paramètres d'un transfert DMU une zone de réécriture en mémoire C-SRAM, au sein de laquelle les données lues depuis la mémoire DRAM seront réécrites et réorganisées. Cette zone de réécriture en mémoire C-SRAM est définie par sa rangée de base.

Nous ajoutons également un **pas de réécriture** des données lues depuis la mémoire DRAM vers la mémoire C-SRAM. L'intérêt de ce pas de réécriture est d'ajouter de la flexibilité dans la réorganisation des données transférées. Un exemple concret d'utilisation de ce pas de réécriture est la réécriture de données au sein d'une zone C-SRAM préalablement initialisée à zéro pour effectuer de l'extension non-signée des données.

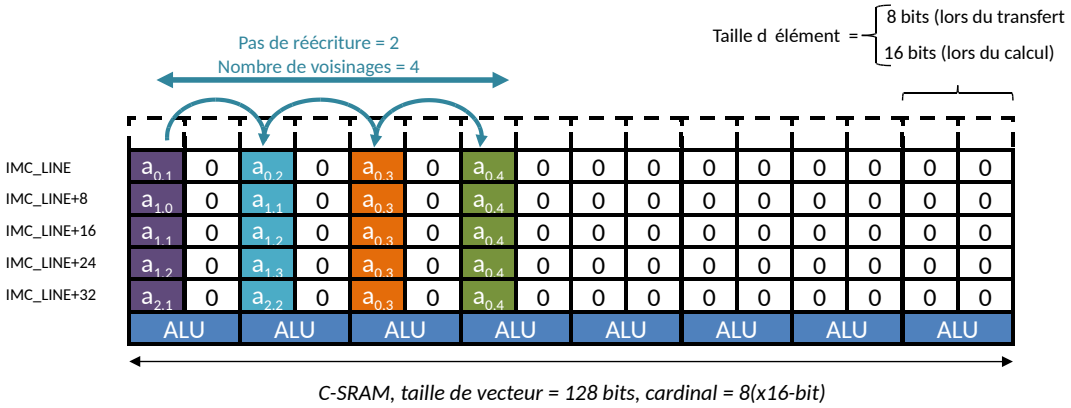


FIGURE 4.3 – Deuxième partie des paramètres du transfert de données présenté lors de la Figure 4.1 et 4.2. Les croix de stencil d'éléments 8 bits sont réécrites verticalement au sein de la mémoire C-SRAM avec un pas de réécriture de deux éléments, pour effectuer de l'extension non signée.

La Figure 4.3 présente l'organisation de données au sein de la mémoire C-SRAM, au terme du transfert paramétré lors des Figures 4.1 et 4.2. Toujours dans cet exemple, la zone de réécriture C-SRAM passée en paramètre possède la rangée de base de valeur `IMC_LINE`. Le pas de réécriture est paramétré à 2, ce qui permet une extension des données transférées de 8 à 16 bits. Cette extension de signe sera utile pour que le HPU puisse programmer des opérations arithmétiques C-SRAM sur ces données, avec moins de risques de dépassement de capacité. Les données de chaque voisinage de stencil sont organisées de telle sorte à être alignées verticalement.

Nous rappelons que la C-SRAM, au temps de la rédaction de la présente thèse, ne contient pas d'opérateurs qui permettent la réduction *horizontale* de données stockées au sein d'un même vecteur. Cette organisation verticale de données permet donc au HPU de vectoriser le calcul de codes de stencil sur plusieurs voisinages de stencil, en opposition à la parallélisation du noyau interne de codes de stencil comme le permettent les opérations de réduction horizontale.

Opération	Paramètres
SET_IN_REGION	Adresse base de la structure de données, taille des rangées au sein de la structure, taille des éléments.
SET_OUT_REGION	Adresse base de la structure de données, taille des rangées au sein de la structure, taille des éléments.
READ_TRANSFER	Coordonnées du point de départ de la lecture `(I,J)`, zone de réécriture en C-SRAM, stride de lecture, stride de réécriture, Géométrie du voisinage de stencil.
WRITE_TRANSFER	Coordonnées du point de départ de la réécriture `(I,J)`, zone de lecture en C-SRAM, stride de lecture, stride de réécriture, Géométrie du voisinage de stencil.
WAIT	Aucun.

TABLE 4.1 – Liste des opérations DMU et leurs paramètres nécessaires pour pouvoir orchestrer le transfert des données de stencils.

Nous établissons ainsi les paramètres suivants pour les transferts de données de stencils de la mémoire principale vers la C-SRAM :

— Les coordonnées du point de départ pour la lecture des données, `(I,J)`.

— La zone de réécriture des données en C-SRAM, `IMC_LINE`.

— La géométrie des voisinages de stencil à transférer.

— Le nombre de voisinages à transférer.

— La taille des données au sein des stencils à transférer.

— Le stride de lecture depuis la structure de données d'entrée, en mémoire principale.

— Le stride d'écriture vers la zone de destination, en C-SRAM.

Pour minimiser le nombre d'arguments nécessaire à l'opération de transfert de stencils vers la C-SRAM – que nous appelleront `READ` par simplicité, nous définissons une opération `SET_IN_REGION` dont le but est de préparer en avance une partie des paramètres constants nécessaires pour `READ`. De façon orthogonale, nous définissons l'opération `WRITE` qui permet de transférer les résultats des codes de stencils vers une structure de données de sortie, paramétrée par une opération `SET_OUT_REGION`. Parce que les codes de stencils sont des opérations de réduction, il n'est donc pas nécessaire que l'opération `WRITE` puisse être paramétrée pour transférer des données selon des voisinages de stencils particuliers. Ces opérations s'exécutent de façon non-bloquante vis-à-vis du HPU, nous définissons donc également une instruction `WAIT` afin de permettre au HPU de se synchroniser avec le DMU lors de la complétion d'un transfert en cours.

Une fois ces opérations définies, nous utilisons les observations suivantes pour les dimensionner au sein du jeu d'instructions de la C-SRAM.

- Le jeu d'instructions doit contenir toutes les opérations DMU définies lors de la précédente sous-section (voir Table 4.1). Nous ne nous imposons, cependant, aucune contrainte concernant le nombre d'instructions nécessaire par opération pour être programmée. Par exemple, utiliser plusieurs instructions pour encoder une opération permettrait d'encoder les paramètres sur une plus large plage de représentation, au détriment d'une latence de programmation plus élevée. Au contraire, utiliser un nombre réduit d'instructions par opérations diminue la latence de programmation mais limite les dimensions des structures de données visées par le DMU, du point de vue des programmeurs. Notons dès maintenant que cet aspect du dimensionnement, comme tous les suivants, ne pourra être tranché que par les besoins applicatifs des logiciels visés par notre implémentation du DMU.

- Le jeu d'instructions *dans son intégralité* doit contenir l'ensemble des paramètres définis lors de la précédente sous-section (voir Table 4.1). Cette contrainte provient du fait que ces paramètres sont définis comme essentiels à la programmation de transferts de stencils. Nous ne nous imposons, cependant, aucune contrainte concernant le positionnement de ces paramètres parmi les instructions implémentant les opérations dont les paramètres sont directement impliqués dans le transfert – c'est-à-dire les opérations `READ`, `WRITE` et `SET_{IN|OUT}_REGION`. L'instruction `WAIT` ne peut pas être utilisée car définie comme utilisée par le HPU pour se synchroniser avec le DMU lors d'un transfert de données de stencils en cours, et donc déjà programmé.

- Les références vers les zones mémoires C-SRAM, source comme destination doivent être encodées sur 16 bits afin d'être orthogonal avec la plage d'adressage de rangées mémoire offertes par le jeu d'instructions de la C-SRAM.

- Les adresses de référence des structures de données stockées en mémoire virtuelle et visées par le DMU doivent être encodées sur 32 bits. La raison pour cette contrainte est que la C-SRAM, en état, a été étudiée et implémentée pour des systèmes 32-bit. Et bien qu'il soit possible de réduire la plage de représentation des adresses en restreignant les structures de données visées à celles stockées en mémoire principale – ce qui permettrait d'en utiliser les adresses physiques, nous souhaitons laisser la possibilité aux programmeurs de viser également d'autres zones mémoires virtuelles dédiées à des périphériques telles que des capteurs photo et vidéo.

- Les paramètres dimensionnant les structures de données stockées en mémoire principale doivent être de taille égale, par souci de consistance. Cela implique, par exemple, que la taille du paramètre *Width* des opérations `SET_{IN|OUT}_REGION` doit être égale à celle du paramètre *Length* des instructions `READ` et `WRITE`. De façon similaire, les références vers les points de départ des transferts de données de stencils – c'est-à-dire au moins le paramètre `REGION.J` – doivent être dimensionnées selon les dimensions des structures de données visées.

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
SETR SETW	1	0	0	0	0	0	Op. code								DATA SIZE			REGION.WIDTH														0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	REGION.BASE_ADDR																															
READ0 WRITE0	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	1	0	0	0	0	0	Op. code								CSRAM.LINENO																0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	REGION.I													0	0	0	REGION.J												
READ1 WRITE1	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	1	0	0	0	0	0	Op. code								0	0	0	TRANSFER.LEN													0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	SRC.STRIDE														DST.STRIDE												NEIGHBOUR -HOOD ID					
WAIT	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	1	0	0	0	0	0	Op. code								Undefined																0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Undefined																															

TABLE 4.2 – L'encodage des instructions DMU se calque sur celui des instructions C-SRAM pour en permettre l'intégration en tant qu'extension de jeu d'instructions.

La Table 4.2 présente notre dimensionnement du jeu d'instructions du DMU à partir des contraintes précédemment énoncées. Les instructions `SETR` et `SETW` implémentent respectivement les opérations `SET_{IN|OUT}_REGION`, et permettent le paramétrage de structures de données en mémoire virtuelle larges de 2^{13} = 8 Kilo-éléments, c'est-à-dire avec des rangées maximales de taille variant entre 8 Kilo-octets, 16 Kilo-octets et 32 Kilo-octets selon la taille de donnée paramétrée (8-bit, 16-bit ou 32-bit). De telles largeurs nous permettent de couvrir la majorité des tailles de structures de données définies par les résolutions d'écran et de capteurs photo/vidéo. Les opérations `READ` et `WRITE` sont décomposées en conséquence en deux instructions pour avoir la place d'encoder les paramètres selon les contraintes précédemment énoncées.

Au terme de ce dimensionnement, nous réservons 5 bits pour encoder le voisinage selon lequel récupérer les données de stencil, une taille de données limitée pour encoder une telle information. Notre contribution pour répondre à cette problématique est un mécanisme de microcode spécifique au DMU pour encoder des transferts de données selon des géométries de stencil.

4.2.1 Microcode DMU pour le transfert de données de stencil

Nous présentons dans cette sous-section notre microcode pour encoder des voisinages de stencil.

Le format de ce microcode permet l'encodage de géométries arbitraires de dimensions maximales 8×8 sur 64 bits. Chaque bit du microcode correspond au *canevas* du voisinage de stencil, ou l'espace sur lequel la géométrie du voisinage de stencil est représentée. Au sein de ce canevas, les géométries de voisinages de stencil peuvent être programmées en mettant à 1 les bits nécessaires pour les encoder. Le bit 36 représente le centre du voisinage de stencil.

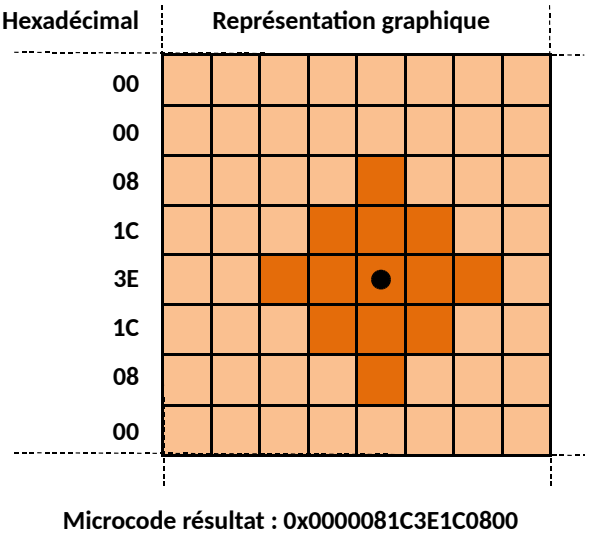


FIGURE 4.4 – Encodage d'un stencil en forme de diamant 5×5, au sein du canevas 8×8 décrit par le format du microcode DMU.

La Figure 4.4 présente un exemple de micro-instruction DMU, qui encode un voisinage de stencil en forme de diamant 5×5. Nous représentons à gauche de la figure l'encodage du voisinage en hexadécimal, et à droite sa représentation graphique au sein du canevas définit par le format de notre microcode DMU.

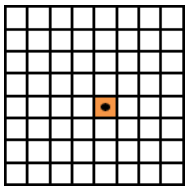
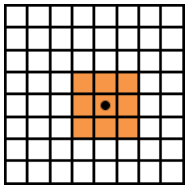
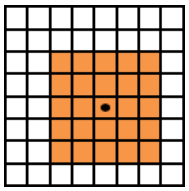
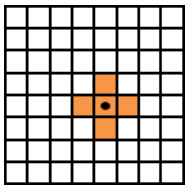
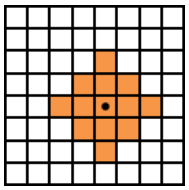
Voisinage	**Microcode**	**Représentation gaphique**	**Usages**
Identité	`0x0000000008000000` `0x0000000000000000`		Transferts en bloc
Carré 3×3	`0x0000001C1C1C0000`		Traitement d'image, Vision par ordinateur
Carré 5×5	`0x00003E3E3E3E3E00`		Traitement d'image, Vision par ordinateur
Croix de *Von Neumann* à 5 points	`0x000000081C080000`		Solveurs linéaires
Diamant 5×5	`0x0000081C3E1C0800`		Dématriçage discret

TABLE 4.3 – Une grande proportion des géométries de stencils les plus courantes peut être encodée dans notre format de microcode.

Pour démontrer que notre microcode DMU est adapté à nos domaines scientifiques d'intérêt, nous présentons dans la Table 4.3 un ensemble de géométries de voisinages de stencil dont les dimensions tiennent intégralement dans le canevas décrit par le format du microcode. Ces différentes géométries sont employées dans les domaines du traitement d'image et de la vision

par ordinateur.

4.3 Fonctionalité du DMU

4.3.1 Exemple de fonctionnement du DMU

Pour présenter les transferts de données générés par le DMU à partir de son jeu d'instructions et de son microcode, nous représentons de façon graphique des exemples de transferts de données. Le premier exemple effectue le transfert d'un voisinage en forme de croix à 5 points, et le second de quatre voisinages de la même forme.

Ces croix de stencils sont transférées de la DRAM vers la C-SRAM à un stride de lecture d'un élément, et un stride de réécriture de deux éléments. Le pas de lecture de ces transferts entraîne une redondance de données entre voisinages de stencil consécutifs, et nous implémentons dans le DMU un mécanisme pour identifier ces données redondantes et les temporiser. Les Figures 4.2 et 4.3 représentent les organisations des données accédés en lecture depuis la DRAM et réécrites dans le C-SRAM pour le second cas.

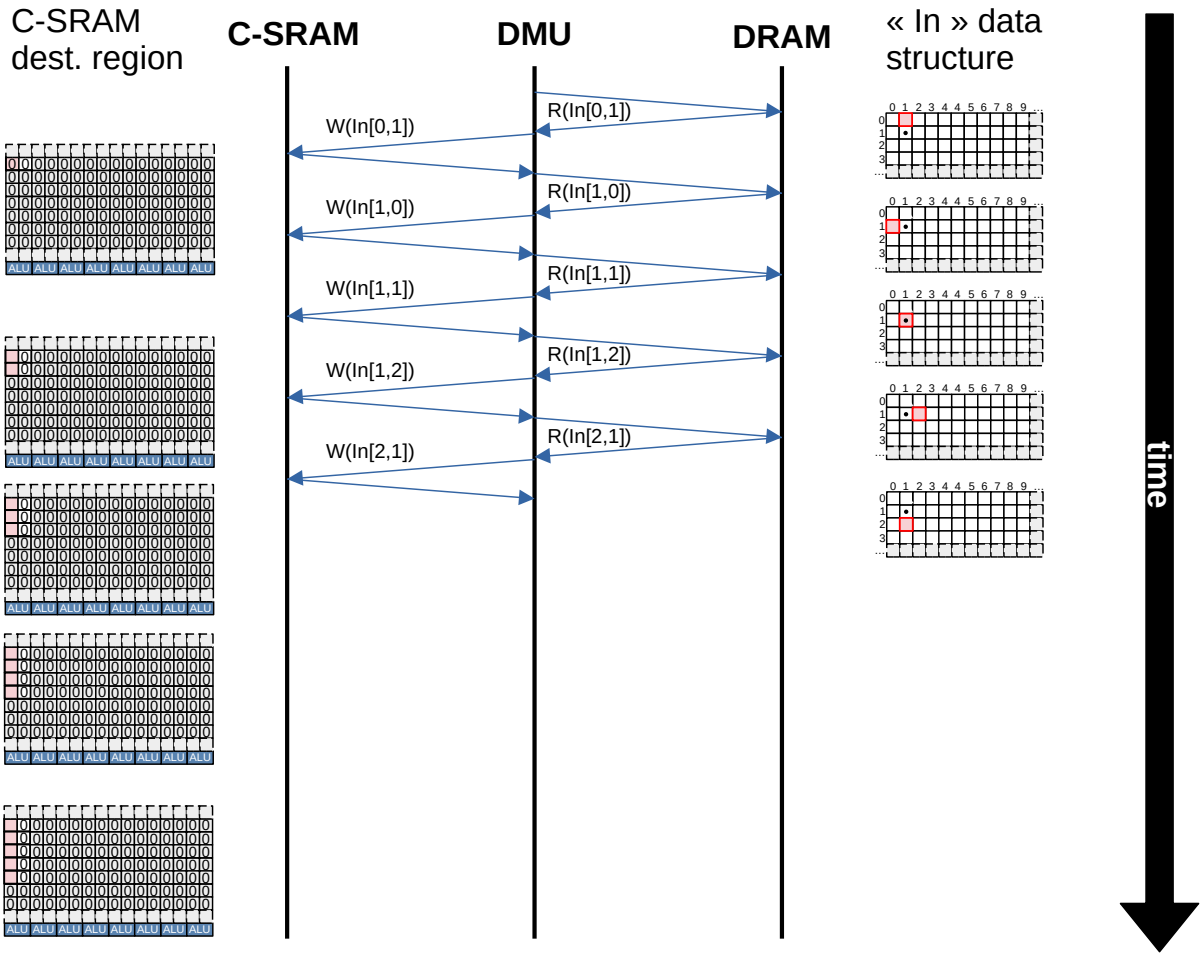


FIGURE 4.5 – Exemple du transfert d'une croix de stencil de la DRAM vers la C-SRAM, avec le DMU.

La Figure 4.5 représente les opérations mémoires effectuées par le DMU au cours du temps pour le transfert d'une croix de stencil, à partir du point de départ de coordonnées (1,1). Ce point de départ correspond au centre du voisinage de stencil, et est utilisé pour identifier les données à transférer. Du fait qu'un seul voisinage de stencil est transféré, il n'existe aucune opportunité à saisir par le DMU quant à la redondance des données lues.

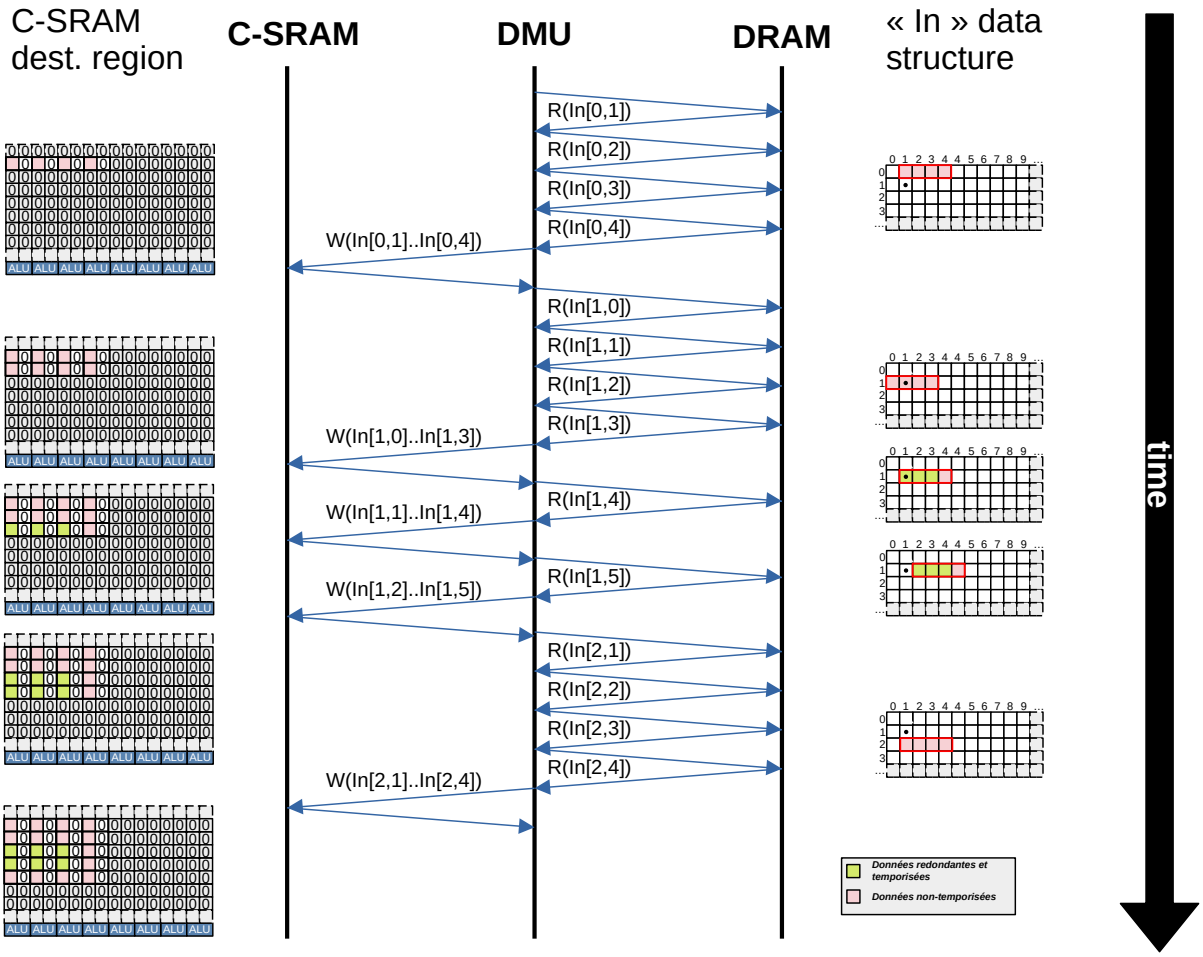


FIGURE 4.6 – Exemple du transfert de quatre croix de stencil de la DRAM vers la C-SRAM, avec le DMU. Les données redondantes entre plusieurs voisinages de stencil ne sont lues qu'une seule fois depuis la DRAM pour optimiser la latence globale du transfert mémoire.

La Figure 4.6 représente les opérations mémoires effectuées par le DMU au cours du temps pour le transfert de quatre croix de stencil, à partir du point de départ de coordonnées (1,1). Les points ((1,1),(1,2),(1,3),(1,4)) correspondent aux centres des voisinages de stencil à transférer, et sont identifiés par le point de départ paramétré pour le transfert.

Le pas de lecture est égal à 1, et il y a donc une redondance de données sur les points de gauche, du centre, et de droite de chaque voisinage de stencil transféré. Dans le cas du transfert de données présenté en exemple :

— Les points ((0,1),(0,2),(0,3),(0,4),(1,0),(1,5),(2,1),(2,2),(2,3),(2,4)) ne sont pas redondants.

— Les points ((1,1),(1,4)) sont redondants à deux des quatre voisinages de stencil à transférer.

— Les points ((1,2),(1,3)) sont redondants à trois des quatre voisinages de stencil à transférer.

Le DMU va temporiser les points ((1,1),(1,2),(1,3),(1,4)) et les réutiliser localement lors de la réécriture en C-SRAM, au lieu de les relire depuis la DRAM. Ainsi, les transferts de données

de stencil sont optimisés de façon déterministe sans intervention des programmeurs.

4.3.2 Intégration du DMU au sein du système

Nous définissons dans cette sous-section les interfaces qui permettent l'interaction entre le DMU et le reste de son système hôte. Rappelons que le DMU est défini comme un contrôleur mémoire avec une ligne de transfert dédiée entre la C-SRAM et la mémoire principale. Il est donc nécessaire d'implémenter des mécanismes d'arbitrage entre les requêtes émises par le HPU et le DMU.

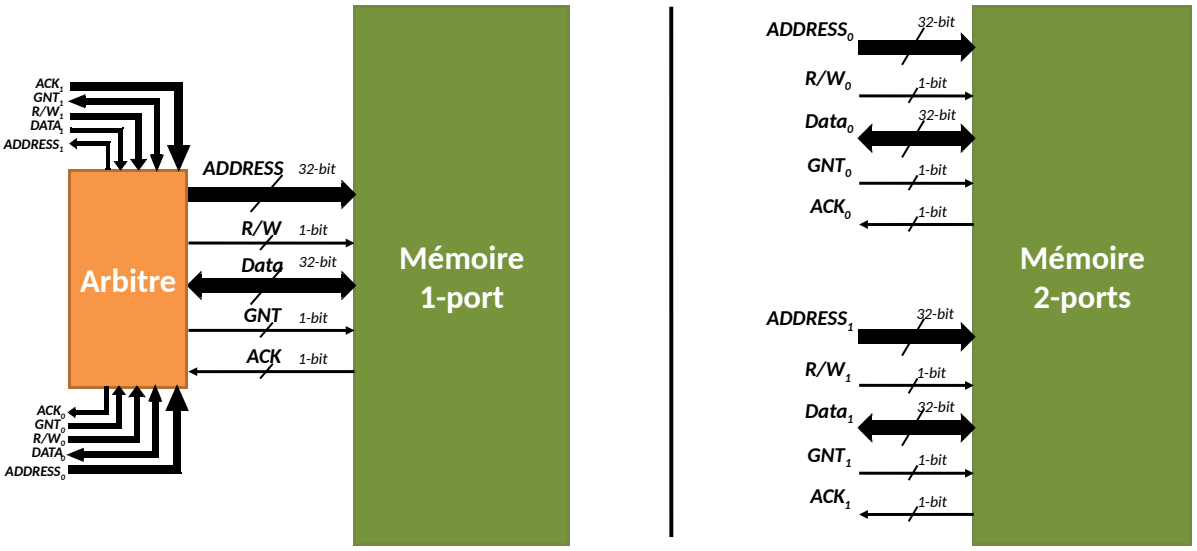


FIGURE 4.7 – L'utilisation de mémoires 2-port permet la réception de requêtes concurrentes de données émises par le DMU et le HPU. Dans le cas contraire un arbitre doit être intégré pour prioriser les requêtes émises par le HPU.

4.4 Implémentation du contrôleur DMU

Lors de la précédente section, nous avons présenté les spécifications essentielles du contrôleur DMU, pour finir sur la définition des opérations nécessaires à la programmation de transferts de données de stencils. Nous proposons donc une implémentation du DMU, dédiée à être fortement couplée avec une mémoire C-SRAM.

4.4.1 Utilisation du microcode au sein du contrôleur DMU

Nous utilisons notre format de microcode au sein du contrôleur DMU pour permettre une flexibilité de programmation de stencils malgré les limitations de taille du champ correspondant au voisinage au sein de l'opération `READ`. Nous ajoutons au contrôleur DMU une mémoire d'un Kilo-bits, associée à un segment de la mémoire virtuelle afin que le HPU puisse programmer à l'avance les voisinages de stencil nécessaires lors de l'exécution des applications. Cette mémoire

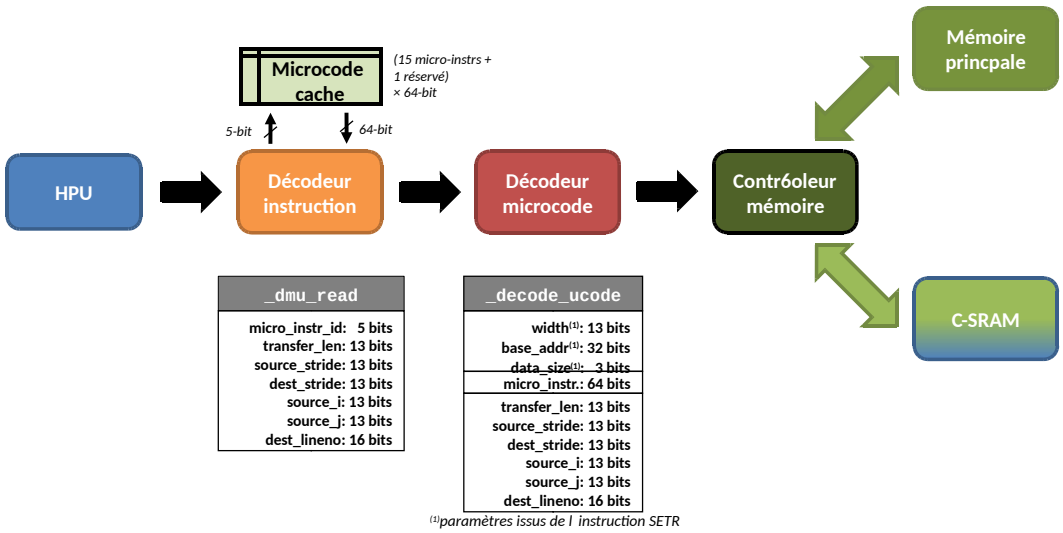


FIGURE 4.8 – Opérations et paramètres internes au contrôleur DMU qui sont impliqués dans l'opération READ.

de microcode peut stocker 16 micro-instructions, pour une capacité totale d'un Kilo-bits. Nous pensons que le surcoût de cette mémoire est raisonnable comparé à d'autres solutions de l'État de l'art mentionné lors du Chapitre 3.

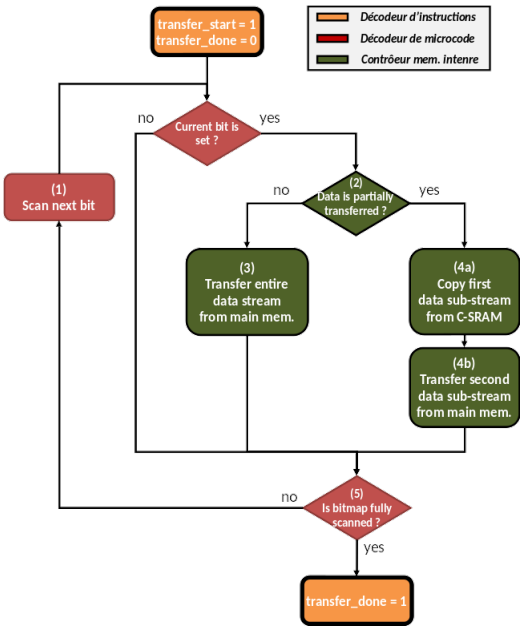


FIGURE 4.9 – Diagramme de fonctionnement du contrôleur DMU lors de l'exécution de l'opération READ. Le code couleur est en relation avec les composants présentés à la Figure 4.8

.

Lors de l'exécution de la routine interne `_decode_ucode`, le décodeur va utiliser les bits du voisinage de stencil encodé pour le transfert afin de générer et mettre à jour des métadonnées pour identifier les données de stencils déjà présents en C-SRAM (Figure 4.9). Ces métadonnées sont obtenues de façon déterministe grâce aux entrées de la routine, et ne reposent sur aucune méthode heuristique pour déterminer des motifs d'accès, au contraire des contrôleurs de cache.

Conclusion

Nous avons présenté le *Data-locality Management Unit* (DMU), un contrôleur mémoire capable de transférer et réorganiser des voisinages de stencils de la mémoire virtuelle vers la mémoire principale. Nous avons défini des spécifications à partir desquelles nous avons réalisé une implémentation fournissant un jeu d'instructions, de telle sorte qu'il puisse être intégré au jeu d'instructions de la C-SRAM en tant qu'instructions de transferts de données. Nous avons mis en lumière des problèmes de dimensionnement du jeu d'instructions, liés à des contraintes architecturales et applicatives et proposé en réponse un format de microcode pour encoder des voisinages de stencils sous format binaire afin de programmer le contrôleur DMU.

Afin d'efficacement programmer des architectures contenant de la mémoire C-SRAM et un contrôleur DMU, il est nécessaire de mettre un point un modèle de programmation permettant de programmer ces derniers. Nous proposons dans le chapitre suivant un modèle de programmation pour résoudre cette problématique.

Chapitre

5 Support logiciel à haut niveau d'une architecture de calcul proche-mémoire

- *Comment décrire à haut-niveau des applications employant des codes de stencil ?*
- *Comment programmer la mémoire C-SRAM et le contrôleur DMU pour exécuter ces applications ?*

Introduction

Nous présentons IMCCCC, un modèle de programmation pour la programmation de la mémoire C-SRAM et du DMU. IMCCC prend la forme d'un DSL basé sur le langage C, avec des types de données spécialisés pour décrire des codes implicitement vectorisés, avec des optimisations dédiées pour les codes de stencil. Le développement de ce compilateur permet ainsi de joindre les contributions applicatives et matérielles présentées jusqu'à maintenant dans le cadre d'un effort de co-conception pluridisciplinaire. Nous proposons également en seconde contribution de ce chapitre une méthodologie d'inter-opération entre IMCCC et Hybrogen, afin de pouvoir générer du code dynamiquement compilé pour allonger le temps d'optimisation des applications au temps d'exécution.

Le chapitre se décompose comme suit : section 5.1 introduit les spécifications générales ainsi que les choix de conception du modèle de programmation. Section 5.2 présente une implémentation d'*imccc* sur la base d'un langage de programmation, notamment le langage C. Section 5.3 étudie l'interopérabilité du modèle de programmation en proposant une méthodologie pour l'implémentation d'applications compilées dynamiquement à partir de code IMCCC, par le biais de Hybrogen. Enfin, Section 5.3 conclut le chapitre.

5.1 Spécification du modèle de programmation IMCCC

```
1  void laplacian(char **In, char **Out, int width, int height)
2  {
3      #pragma imc set ncontexts 1
4      // Stencil coefficients (CPU / IMC)
5      imc_fixed_short coef[] = {1, 1, -4, 1, 1};
6      // Load/Store recipients (CPU / IMC)
7      imc_short stenc[5];
8      imc_short res;
9      // Loop variables (CPU only)
10     int i, j, k;
11     #pragma imc parallel for out
12     for (i = 1; i < height-1; i += 1)
13     {
14         #pragma imc parallel for in
15         for (j = 1; j < width-1; j = j + 1)
16         {
17             /* Input transfer */
18             stenc[0] = In[i-1][j  ];
19             stenc[1] = In[i  ][j-1];
20             stenc[2] = In[i  ][j  ];
21             stenc[3] = In[i  ][j+1];
22             stenc[4] = In[i+1][j  ];
23             /* Computation */
24             res = 0;
25             for (k = 0; k < 5; ++k)
26             {
27                 res += coef[k] * stenc[k];
28             }
29             /* Output transfer */
30             Out[i][j] = res;
31         }
32     }
33 }
```

FIGURE 5.1 – Implémentation d'un opérateur laplacien discret par le biais d'IMCCC, sur la base du langage C.

La Figure 5.1 présente un code écrit en IMCCC. Le code en question est une implémentation d'un opérateur laplacien discret à cinq points[Opea].

IMCCC est paramétré par l'utilisateur par un ensemble de directives de compilation, sur l'inspiration de DSLs tels que l'ISPC d'Intel. Par la nature vectorielle des opérateurs arithmétiques de la mémoire C-SRAM, le modèle de programmation décrit par IMCCC est *implicitement vectorisé*. Nous utilisons ce terme pour opposer IMCCC face à la vectorisation automatique, qui ne nécessite aucune intervention ou connaissance des programmeurs et emploie des heuristiques pour optimiser au mieux un code d'entrée arbitraire de façon transparente. Cette terminologie permet également de ne pas confondre notre modèle de programmation parmi les modèles de programmation *parallèles*, qui permettent du traitement multiprocessus voire multithreads, et non seulement vectorisé.

En utilisant un modèle de programmation implicitement vectorisé, les programmeurs peuvent exprimer des transferts de données de stencils sous une forme proche de la description algorithmique, et donc sans avoir besoin de moyens d'expression spécialisés.

5.1.1 Support syntaxique des ressources C-SRAM

Nous faisons le choix de baser notre modèle de programmation sur la base d'un langage de programmation dit *impératif* pour plusieurs raisons. Premièrement, ces langages de programmations décrivent des algorithmes destinés à être exécutés de façon séquentielle par une machine abstraite quasi équivalente à une machine de Turing. Ce modèle de machine abstraite correspond bien à notre architecture d'intérêt, avec le HPU qui est seul responsable du flot d'instructions et la C-SRAM qui y est interfacée de façon fortement couplée. De plus, les langages de programmation impératifs sont également les langages au paradigme de programmation le plus populaire parmi les développeurs.

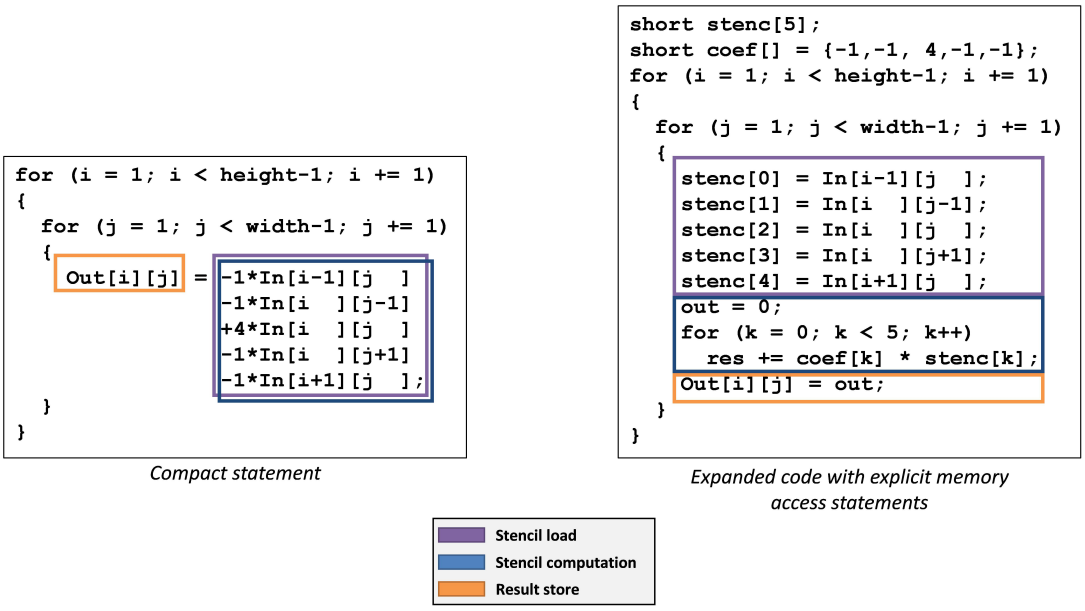


FIGURE 5.2 – Formes usuelles d'expression d'un code de stencil dans un langage impératif, ici un opérateur laplacien discret implémenté en langage C.

Dans la majorité des langages de programmation dits *impératifs*, les codes de stencils sont implémentés par le biais de boucles, potentiellement imbriquées, qui contrôlent les transferts des données de stencils. Usuellement, ce genre de codes va être écrit de façon dite *compacte*, c'est avec plusieurs opérations figurant toutes dans la même clause, ce qui va avoir pour effet de rendre implicite des opérations telles que les transferts mémoires et leur stockage. Notre modèle de programmation requiert des programmeurs de séparer les clauses dédiées au transfert des données de stencils de celles dédiées aux opérations arithmétiques. La Figure 5.2 présente les expressions compactes et étendues du calcul de l'opérateur laplacien dans le code présenté au

sein de la Figure 5.1. Le code étendu explicite le stockage des données lues depuis, et écrites vers la mémoire.

Pour optimiser les imbrications de boucles qui correspondent à des codes de stencil, nous définissons donc des directives spéciales qui les labélisent comme telles.

Directive	Definition
`imc parallel for in`	La boucle annotée est définie comme boucle la plus interne d'un segment de code de stencil.
`imc parallel for out`	La boucle annotée est définie comme boucle la plus externe d'un segment de code de stencil.

TABLE 5.1 – Liste des directives définies par IMCCC qui permettent de paramétrer des segments de code de stencils.

La Table 5.1 présente la liste de ces directives de labélisation. Les directive `parallel for in` et `parallel for out` labélisent respectivement les boucles associés à l'idexation de la dimension 1 et 2 des algorithmes de stencil. Cette labélisation permet de demander au compilateur d'optimiser au sein de ces boucles les transferts de données à l'échelle de ces deux dimensions, ce qui se traduit par l'utilisation automatique du DMU. Ces directives sont également utilisées par le compilateur pour diriger des optimisations de code telles que la vectorisation de boucles. Nous présenterons en plus grand détail le fonctionnement de ces optimisations au cours du chapitre.

Pour apporter aux programmeurs le support explicite des opérations C-SRAM, nous implémentons une nouvelle catégorie de variables scalaires, les variables *IMC* stockées au sein de la C-SRAM.

Nous avons choisi de ne pas effectuer de la vectorisation automatique de code pour deux raisons. Premièrement, elle rend transparentes de l'utilisateur les opérations qui sont effectuées en C-SRAM, ce qui rend difficile les phases de débogage tout autant que celles de programmation. Deuxièmement, la vectorisation automatique pour des architectures hétérogènes telles que notre architecture d'intérêt ajoute énormément de pression sur le compilateur, qui doit maintenant disposer de techniques d'optimisation complexes pour identifier les opérations arithmétiques qui peuvent ou devraient être déchargées vers la C-SRAM a contrario du HPU. En effet, des solutions telles que CAIRO[HNKK17] ou PRIMO[ASL+19], utilisent des heuristiques basées sur des modèles architecturaux dont la complexité est à l'échelle des architectures ciblées, et peuvent présenter des problèmes dont la solution est très complexe à implémenter tout autant qu'à résoudre.

Data size	Data type (fixed context)	Data type (non-fixed contexts)
8	`imc_fixed_char`	`imc_char`
16	`imc_fixed_short`	`imc_short`
32	`imc_fixed_int`	`imc_int`

TABLE 5.2 – Types de données fournis par IMCCC pour manipuler des données en C-SRAM.

La Table 5.2 présentent les types de données supportées par notre modèle de programmation et la C-SRAM. Ces types sont scalaires et permettent d'utiliser la mémoire C-SRAM sans exprimer de façon explicite sa nature vectorielle.

L'utilisation de types de données spécialisés est une approche différente d'interfaces telles qu'OpenMP dans laquelle le code parallélisé est annoté a posteriori de son écriture et ne spécifie pas le rapport de relation entre les nœuds de calcul et les opérations arithmétiques du code. Notre modèle de programmation supporte l'allocation statique et dynamique de ces variables sur une pile stockée dans la C-SRAM.

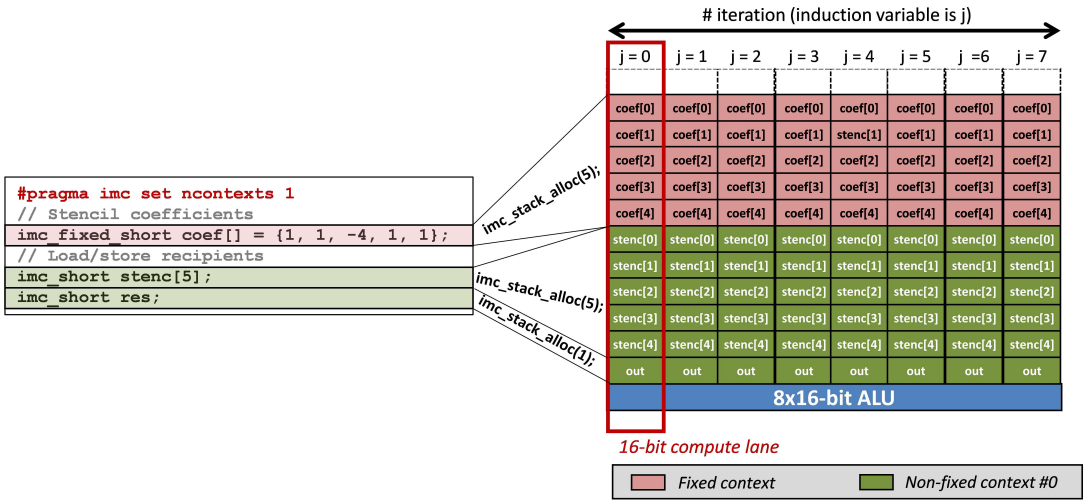
Les variables IMC correspondent à des vecteurs stockés au sein de la mémoire C-SRAM, ce qui veut dire que leur empreinte mémoire est plus grande que leur taille de données intrinsèques. La taille de l'empreinte mémoire des variables IMC est définie par sa taille de données intrinsèque et un paramètre utilisateur que nous appelons le *nombre de contextes logiques* de l'application. Nous définissions les contextes logiques comme des segments de mémoire C-SRAM contenant chacun une pile des variables IMC d'un programme, et dont le nombre influence le degré de vectorisation du code. La notion de contextes logiques selon IMCCC est analogue au concept de *warps* selon Nvidia CUDA ou encore la notion de *gangs* selon ISPC. Nous rappelons cependant qu'a contrario des gangs ou des warps, les contextes IMCCC n'ont qu'une dimension logique et ne transparaissent d'aucune implémentation matérielle particulière. Chaque contexte correspond à une pile de valeurs C-SRAM à des instants de vectorisation de code différents, ce qui signifie que seules les données stockées dans les mêmes contextes logiques peuvent interagir entre elles en temps normal.

Nous introduisons également un contexte spécial, que nous appelons le contexte *figé*. Toute variable déclarée au sein de ce contexte unique peut être partagée auprès des autres contextes normaux. L'ajout du concept de contexte figé permet de déclarer des variables constantes, et qui seraient donc utilisées pour plusieurs contextes normaux. Il est également possible d'utiliser des variables IMC au sein du contexte figé comme interfaces d'échange entre différents contextes, par exemple pour implémenter des réducteurs.

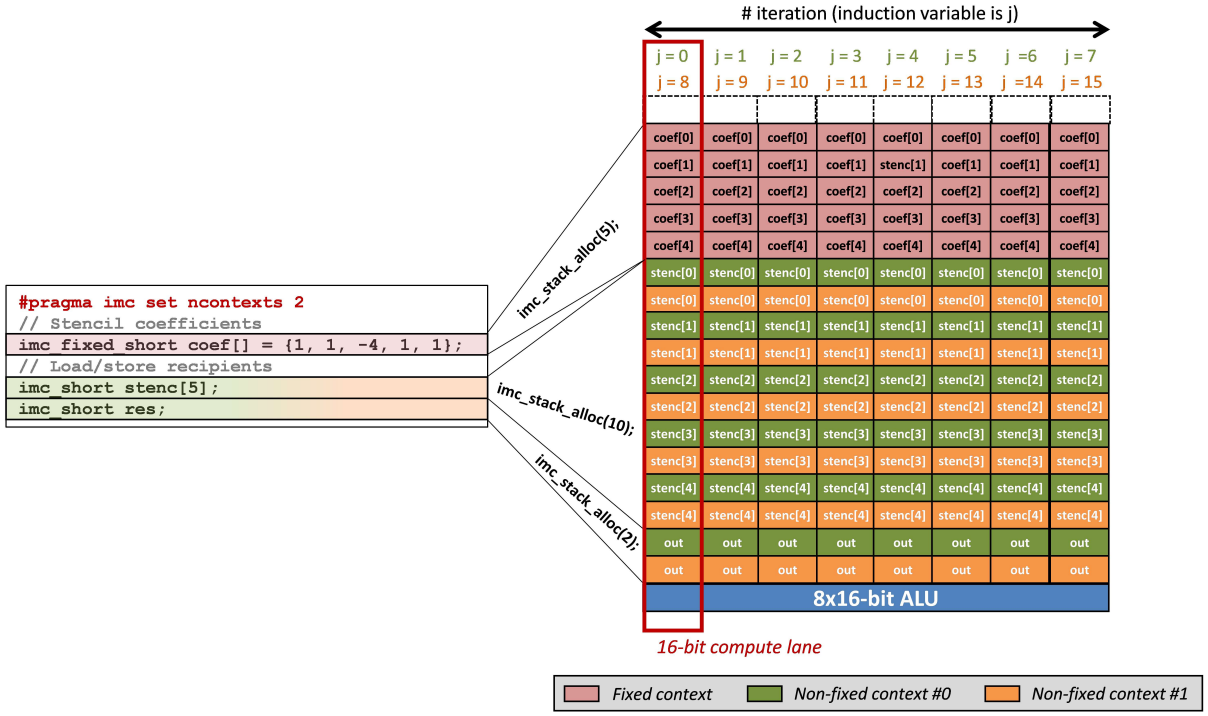
Cette définition se formalise par l'équation suivante (la clause en gras correspond à une clause sémantique et la clause en italique à une clause mathématique) :

$$\textbf{\textit{"dest = src1 op src2"}} \Rightarrow \forall n \in [0; nctx[, dest_n = op(src2_n, src1_n) \tag{5.1}$$

Le nombre de contexte logique sert aux programmeurs à paramétrer de façon implicite l'allocation de la mémoire C-SRAM sans que cette dernière ait un impact sur la description du programme, qui reste scalaire et n'introduit pas de référence à des spécifications architecturales de la mémoire C-SRAM.



(a) Avec un contexte logique.



(b) Avec deux contextes logiques.

FIGURE 5.3 – Les clauses de déclarations de variables IMC peuvent être paramétrées avec des directives `imc` pour augmenter la taille des zones mémoires C-SRAM associées à ces dernières.

Les Figures 5.3a et 5.3b présente l'impact du nombre de contextes logique sur l'implémentation du code présenté lors de la Figure 5.1, en utilisant un nombre de contextes logiques respectivement égal à 1 et 2. Pour chacune des deux figures, le bloc de gauche correspond à la déclaration des variables IMC, et le bloc de droite a l'organisation des données en mémoire C-SRAM. Nous supposons, que la taille des vecteurs C-SRAM est de 128 bits.

Dans les deux cas les coefficients de l'opérateur laplacien, `coef`, sont déclarés comme un tableau de type `imc_fixed_short`, qui contient cinq éléments. L'empreinte de ce tableau en mémoire C-SRAM correspond donc à cinq vecteurs de cardinalité 8, chaque vecteur correspondant à un élément de `coef`.

Les variables `stenc` et `res`, elles, sont déclarées de type `imc_short`, et sont donc allouées dans des contextes logiques. Dans le cas de la Figure 5.3a, le tableau de cinq éléments `stenc` correspond à cinq vecteurs en mémoire C-SRAM, tandis que `res` correspond à un vecteur. Lors de la vectorisation de la boucle annotée avec `imc parallel for in`, le degré de vectorisation sera égal à 8, et lors de l'entrée de la boucle, les vecteurs associés à `stenc` et `out` contiendront les données pour j allant de 0 à 7.

Dans le cas de la Figure 5.3b, le tableau de cinq éléments `stenc` et la variable `out` correspondent respectivement à 10 et 2 vecteurs de cardinalité 8, en mémoire C-SRAM, car le nombre de contextes logiques définis par l'utilisateur est égal à 2. Lors de la vectorisation de la boucle, le degré de vectorisation sera égal à $2 \times 8 = 16$. Lors de l'entrée de la boucle, les vecteurs associés à `stenc` et `out` dans le contexte logique #1 contiendront les données pour j allant de 0 à 7, tandis que les vecteurs du contexte #2 contiendront les données pour j allant de 8 à 15.

Lorsque le nombre de contexte logique est supérieur à 1, ces derniers sont organisés en entrelacement dans la mémoire C-SRAM pour maximiser la localité spatiale des transferts de données, et réduire le nombre d'appels aux instructions DMU lors des transferts de données entre la mémoire C-SRAM et la mémoire DRAM. Nous présenterons dans une section suivante une implémentation de compilateur IMCCC capable de gérer cette spécification, et l'impact de cette dernière sur l'optimisation du code.

Expression synaxtique	Séquence d'opérations C-SRAM	Opérations couvertes
dest = *value*;	bcast dest, value	Assignation de constante à chaque ligne de calcul
dest = *oper1 op oper2*;	op dest, oper2, oper1	Opération binaire sur chaque ligne de calcul ('+', '-', '*')
dest = *oper1 op value*;	op dest, value, oper1	Opération binaire sur chaque ligne de calcul ('«', '»')
dest = *oper1 cond oper2* ? *oper3* : *oper4*;	cmp flag, oper1, oper2 copyXX dest, flag, oper3 copynX dest, flag, oper4	Assignation conditionnelle
dest = *mem_src*[*index*]	DMU transfer (HW or SW-emulated)	Transfert de données (Mem. principale → C-SRAM)
mem_dest[*index*] = *src*	DMU transfer (HW or SW-emulated)	Transfert de données (C-SRAM → Mem. principale)

TABLE 5.3 – Liste des clauses syntaxiques supportées par le modèle de programmation. *dest*, *src*, *oper1* et *oper2* font référence à des variables IMC, tandis que *value*, *mem_dest* et *mem_src* font référence à des variables HPU.

La Table 5.3 présente les différentes clauses supportées par IMCCC pour permettre aux programmeurs l'utilisation des opérations arithmétiques de la C-SRAM. Ces clauses spécifient un fort typage des opérations vis-à-vis des variables IMC et HPU, qui ne peuvent être utilisés dans les mêmes clauses que dans des cas très spécifiques (assignation, transfert en lecture/écriture, opérations de décalage). Grâce à cette spécification, les compilateurs implémentant IMCCC ne requièrent pas des heuristiques dédiées à la vectorisation de code aussi complexes que celles employées pour la vectorisation automatique. Nous utilisons les opérateurs ternaires pour pouvoir effectuer des clauses d'assignation conditionnelle sur des variables IMC sans avoir besoin de faire d'analyse de code complexe et tout en permettant de réserver les blocs de flot de contrôle au HPU.

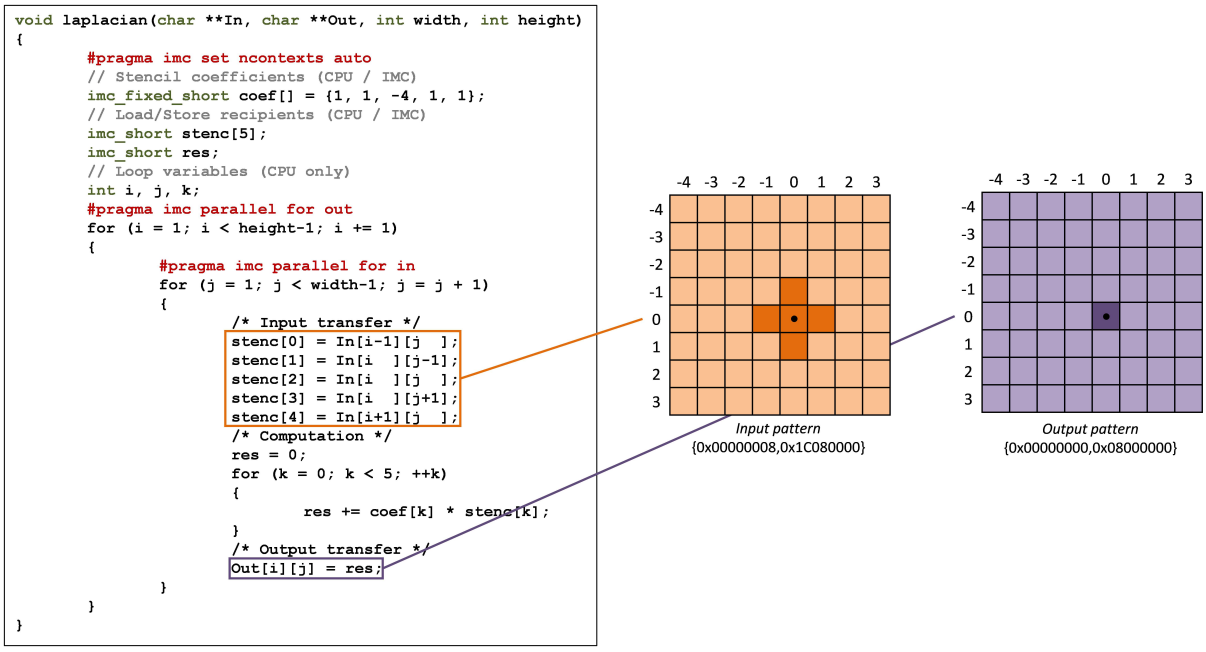


FIGURE 5.4 – Les clauses supportées par le modèle de programmation IMCCC permettent d'exprimer le transfert de données de stencils de façon explicite et pouvant être analysée par un compilateur.

Nous choisissons de spécifier le support des transferts de données de stencils optimisés en utilisant les clauses de lecture de la mémoire principale vers des tableaux IMC comme interface de programmation. Par ce choix de spécification, il devient plus facile pour le compilateur de faire de l'analyse de code pour détecter de façon statique les voisinages de stencils résultants (Figure 5.4). Comme discuté précédemment, les codes de stencils sont intrinsèquement des opérations de réduction au rapport N-vers-1, il ne nous est donc pas nécessaire de faire ce genre d'analyse pour des opérations d'écritures générales.

5.2 Implémentation de IMCCC

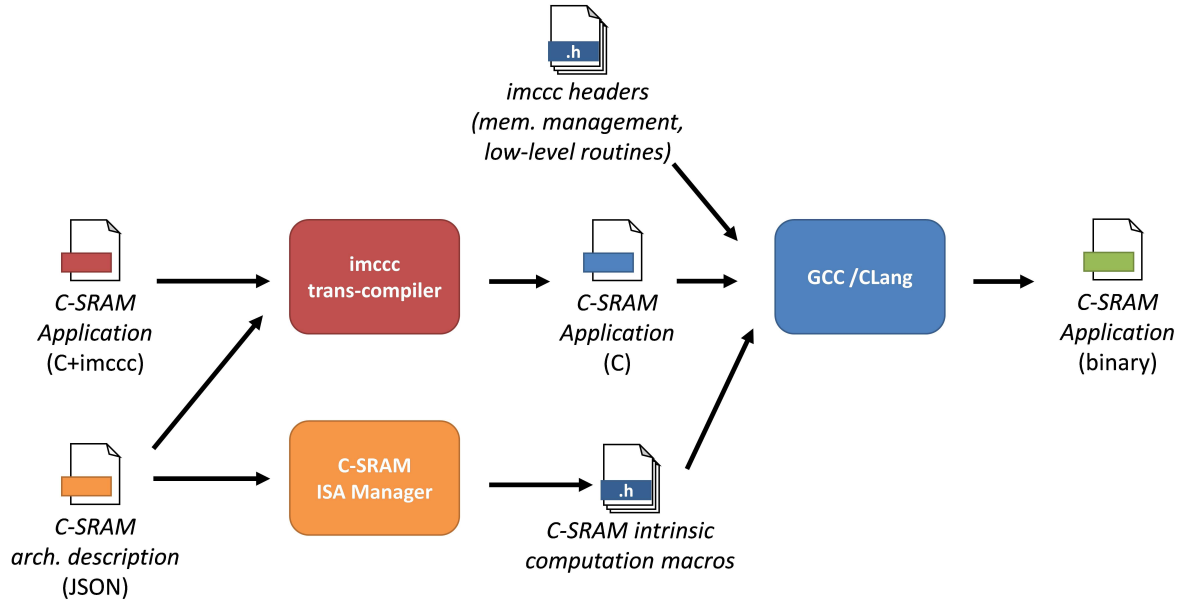


FIGURE 5.5 – 'implémentation du compilateur IMCCC utilise le langage C comme intermédiaire pour la production d'applications dédiées à la C-SRAM et au contrôleur DMU.

Afin d'effectuer des évaluations de notre modèle de programmation, nous faisons l'implémentation d'un compilateur IMCCC. Ce compilateur prend en entrée du code écrit avec le modèle de programmation IMCCC et le langage C, ainsi qu'une description architecturale de la mémoire C-SRAM visée pour dimensionner le code. Notre compilateur produit alors une version optimisée et vectorisée du code, en langage C, qui peut être compilée avec n'importe quel compilateur supportant ce langage. Le flot de compilation repose sur deux jeux d'APIs internes pour implémenter le support de la C-SRAM en langage C pur. La première implémente le support de jeu d'instructions C-SRAM sous la forme de macros, et est générée par le gestionnaire de jeu d'instructions C-SRAM, développé au sein de notre équipe de recherche. La seconde est spécifique à IMCCC et contient les routines bas-niveau dédiées à l'allocation de ressources ainsi que les routines de programmation et d'émulation du contrôleur DMU. Ces deux APIs internes sont générées à partir de la description d'architecture C-SRAM.

5.2.1 Gestion de la mémoire C-SRAM

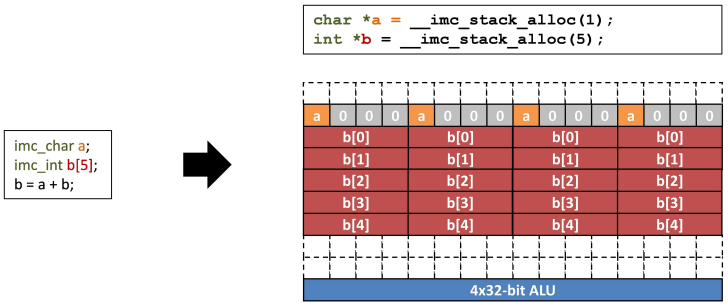


FIGURE 5.6 – L'alignement de variables de tailles de données différentes au sein d'un programme est effectué sur la base du type de donnée de taille la plus grande.

Pour être conforme aux spécifications du modèle de programmation IMCCC, notre compilateur doit assurer que les calculs décrits par le programmeur soient réalisables. Comme discuté lors de précédents chapitres, la bonne réalisation de calculs en mémoire C-SRAM dépend du bon alignement des données concernées. La question de l'alignement de données se soulève particulièrement lorsqu'un programme IMCCC utilise des types de données de tailles différentes, mais impliquées dans les mêmes calculs. Dans cette situation, il est alors nécessaire de faire ce que nous appelons une *conversion de type implicite*, pour que toutes les variables soient de la même taille lors ed leur stockage en mémoire C-SRAM et puissent être calculées. La Figure 5.6 présente cette problématique avec l'utilisation de deux variables : 'a' un élément de 8 bits et 'b' un tableau d'éléments de 32 bits. Notre compilateur effectue la conversion implicite de 'a' de 8 vers 32 bits pour que les deux variables soient bien alignées lors de calculs.

5.2.2 Vectorisation des boucles

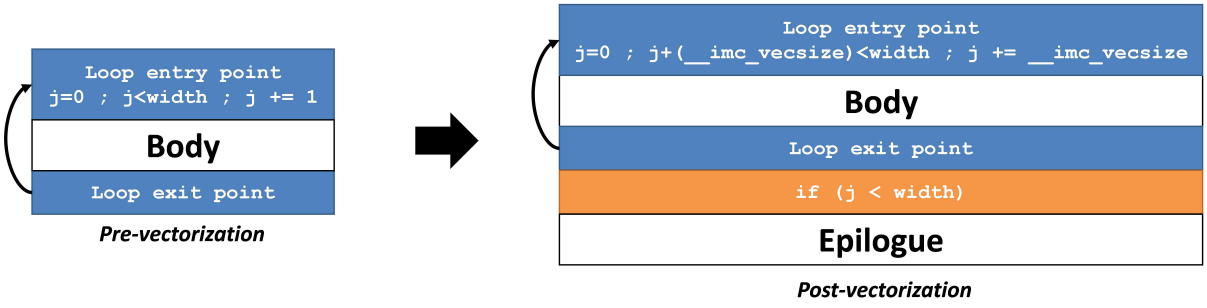


FIGURE 5.7 – Les boucles vectorisées par la directive `imc parallel for in` sont complétées avec un épilogue (bloc `if`) pour effectuer les calculs éventuellement restants tout en évitant des erreurs de segmentation.

Notre compilateur IMCCC vectorise les boucles identifiées par la directive `imc parallel for in` selon un *degré de vectorisation*. Ce degré de vectorisation correspond au produit du

nombre de rangées de calcul par le nombre de contextes. Le premier est paramétré par la taille de données IMC maximale et le second par les directives IMCCC. Pour éviter des risques d'erreur de segmentation ou autres effets de bord indésirables liés à cette optimisation, nous bornons la boucle avec ce que nous appelons un *épilogue*, qui contient une copie du code de la boucle d'origine modifiée pour recalculer les quantités de données et de calculs nécessaires pour finir la boucle. L'épilogue contient notamment en prologue un ensemble de clauses, pour recalculer les longueurs des transferts de données ainsi que le nombre de contextes nécessaires pour les stocker et les calculer.

5.2.3 Détection de voisinages de stencils et gestion des accès mémoire

Rappelons qu'au sein de notre modèle de programmation, un segment de code de stencil donné se définit comme les segments de code contenus dans les boucles annotées à l'aide des directives `imc parallel for out` et `imc parallel for in`. Ces annotations vont nous permettre d'identifier les *variables d'induction du code de stencil*, c'est-à-dire les variables directement responsables du contrôle du code interne au segment de code de stencil. Nous implémentons dans notre compilateur IMCCC un algorithme qui nous permette de détecter de façon statique des voisinages de stencils décrits dans le code, grâce à des informations telles que l'identification des variables d'induction.

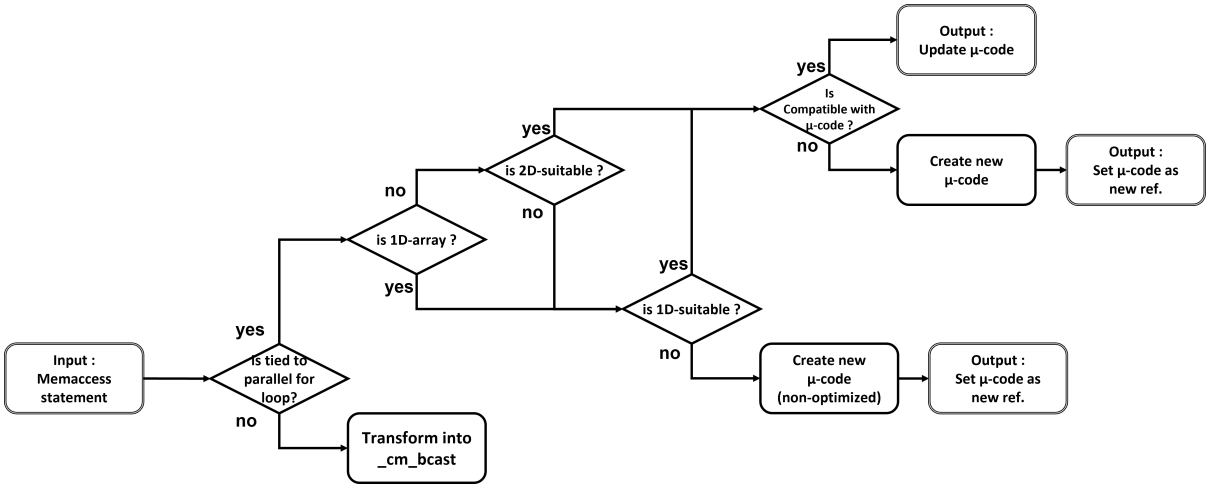


FIGURE 5.8 – Algorithme d'association des clauses d'accès mémoires.

Cet algorithme (Figure 5.8) va analyser le code complet d'entrée pour trier dans un premier temps les clauses d'accès mémoire en deux catégories : celles qui sont associées à des codes de stencils et celles qui ne le sont pas. Les clauses d'accès mémoire associées à des codes de stencils peuvent être détectées par les propriétés suivantes : elles sont localisées au sein d'un segment de code de stencil, et l'indice de dernier ordre de l'opérande doit être dépendant au moins de la variable d'induction interne du segment de code de stencil. Notons également que les dimensions matérielles du DMU limitent également les tailles de voisinages de stencil supportés en fonction de son implémentation. Nous avons défini comme implémentation de référence pour le reste du

présent document un contrôleur DMU pouvant être programmé pour transférer des voisinages de stencils aux dimensions inférieures à 8×8 selon une borne $[-4, 3]$ en fonction du point de référence. Ces observations nous permettent donc d'établir des formes sémantiques normalisées de clauses d'accès mémoires pouvant bénéficier de l'utilisation du contrôleur DMU.

Nous établissons la forme normalisée suivante pour des structures de données au moins à deux dimensions, implémentées avec un niveau d'indirection :

$$\textbf{"O[I}_1\textbf{][I}_0\textbf{]"}, \begin{cases} I_1 \Rightarrow C_1 + ind_1 + x \\ I_0 \Rightarrow C_0 + ind_0 + y \\ -4 \leq x_{\mathbb{Z}} \leq 3 \\ -4 \leq y_{\mathbb{Z}} \leq 3 \\ \{ind_0, ind_1\} \notin (O \bigcup C_1 \bigcup C_0) \end{cases} \tag{5.2}$$

O correspond à l'adresse de base de la structure de données accédée par la clause d'accès mémoire analysée. $_1$ et I_0 correspondent respectivement aux indices d'avant-dernier et de dernier ordre de O. C_1 et C_0 sont appelées les *parts constantes* de I_1 et I_0, il s'agit d'expressions au sein des indices de derniers ordres de O qui n'ont pas d'impact dans la définition du voisinage de stencil selon les dimensions fixées par le contrôleur DMU. ind_1 et ind_0 correspondent respectivement aux variables d'induction interne et externe du segment de code de stencil propriétaire de la clause en cours d'analyse. Enfin, x et y sont les identifiants de la composante du voisinage de stencil en cours de construction. Il est important que ind_1 et ind_0 soient uniques dans l'expression de l'opérande de l'accès mémoire, pour garantir que le voisinage de stencil détecté est consistant d'une itération de boucle à l'autre.

Nous établissons également une forme normalisée pour des structures implémentées sans niveau d'indirection, c'est-à-dire de la forme suivante :

$$\textbf{"O[I]"}, \begin{cases} I \Rightarrow C + (ind_1 + x) \times W + ind_0 + y \\ -4 \leq x_{\mathbb{Z}} \leq 3 \\ -4 \leq y_{\mathbb{Z}} \leq 3 \\ \{ind_0, ind_1\} \notin (O \bigcup C \bigcup W) \end{cases} \tag{5.3}$$

L'expression de I fait référence à un accès à une structure de données stockée sous forme *row-major*, qui correspond à un arrangement contigu des rangées en mémoire. De façon analogue à la forme normalisée précédente, ind_1 et ind_0 doivent être exprimées de façon unique dans l'expression de l'opérande de l'accès mémoire.

```
- ImcFunctionAnalyzer analysis report :
  - Data size for IMC computation : 'imc_fixed_short' (16-bit)
  - context size : 6
  - fixed context size : 5
  - number of contexts : 1
--- Starting DmuStatementBuilder ---
Creating DMU call (In[i][j] --> stenc[0], pat=0x0000000800000000)
Updating DMU call (In[i][j] --> stenc[0], pat=0x0000000810000000)
Updating DMU call (In[i][j] --> stenc[0], pat=0x0000000818000000)
Updating DMU call (In[i][j] --> stenc[0], pat=0x000000081c000000)
Updating DMU call (In[i][j] --> stenc[0], pat=0x000000081c080000)
Creating Bcast call (0 --> res)
Creating DMU call (res --> Out[i][j], pat=0x0000000000000000)
```

FIGURE 5.9 – Classification des clauses d'accès mémoire du code en Figure 5.4. Le voisinage de stencil a bien été détecté pour en construire une micro-instruction DMU.

Ces formes normalisées sont utilisées comme patron pour compacter le plus possible les transferts de données de stencil au sein d'instructions DMU. Les voisinages de stencils établis sont ensuite utilisés pour la programmation des transferts DMU, à l'aide du format de microcode présenté dans le chapitre précédent. La Figure 5.9 présente la passe de notre implémentation sur le code présenté en Figure 5.4. Les métadonnées relatives aux voisinages de stencils sont conservées et mises à jour d'une clause satisfaisante à l'autre. Les clauses correspondant au transfert en écriture des résultats donnent toujours lieu à des transferts unitaires, de par la nature réductrice des codes de stencils. Enfin, les clauses ne dépendant pas des variables d'induction du stencil sont remplacées par des instructions de broadcast pour initialiser les variables de destination.

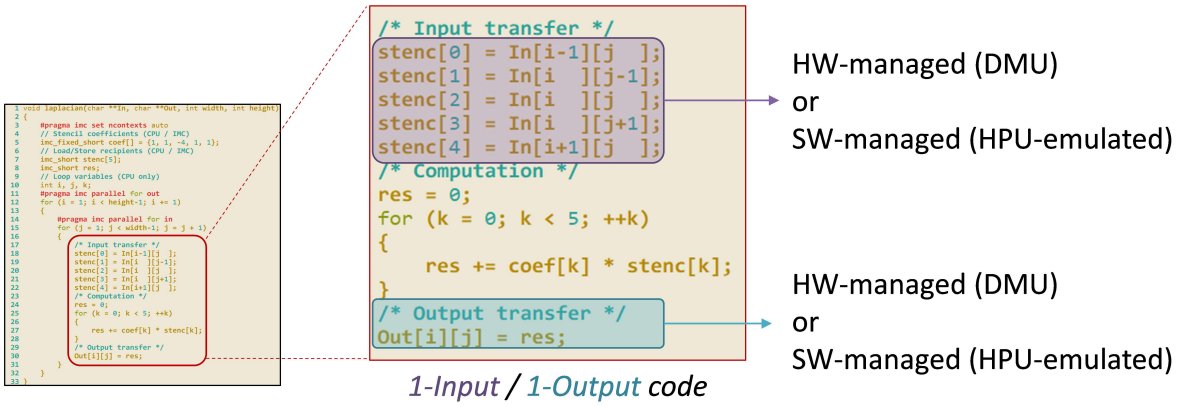


FIGURE 5.10 – Le code de l'opérateur laplacien présenté en exemple possède après analyse un flux d'entrée et un flux de sortie, chacun pouvant être associé à canal de transfert différent.

Les transferts de données de stencils peuvent être paramétrés depuis le compilateur `immcc` pour être déchargées soit auprès du contrôleur DMU matériel, soit auprès du HPU par le biais d'une fonction d'émulation du DMU (Figure 5.10). L'intérêt de permettre ce paramétrage des transferts de données permet dans un premier temps de pouvoir supporter des architectures

intégrant la mémoire C-SRAM sans le DMU. Nous notons également l'intérêt d'étudier de l'utilisation du DMU et des contrôleurs mémoires à disposition du HPU par le biais de sa hiérarchie mémoire pour l'optimisation de transferts de données. Cette étude sera présentée en plus grand détail dans le chapitre suivant.

5.3 Interopérabilité de IMCCC avec Hybrogen pour la compilation dynamique d'applications C-SRAM

Lors du chapitre 4, nous avons présenté Hybrogen, un environnement de programmation dédié à la compilation dynamique de code. Les intérêts de la compilation dynamique sont de permettre de résoudre des problèmes d'optimisations de code autrement insolubles avec la seule utilisation d'informations connues avant l'exécution de l'application. Ce paradigme de génération de code présente un intérêt pour l'optimisation d'applications dont les valeurs ne peuvent être résolues qu'au runtime. Pour cette raison, nous proposons une méthodologie permettant de compiler des applications C-SRAM bénéficiant de la génération dynamique de code, en utilisant Hybrogen comme intermédiaire. Nous spécifions à notre compilateur IMCCC la possibilité de générer du code Hybrolang, le langage propriétaire de Hybrogen, pour interfacer ce dernier avec notre compilateur.

IMCCC	Hybrolang
`imc(_fixed)_char`	`int[] 8 16`
`imc(_fixed)_short`	`int[] 16 8`
`imc(_fixed)_int`	`int[] 32 4`

TABLE 5.4 – Association des types de données IMC du modèle de programmation IMCCC vers Hybrolang

Tout d'abord, nous ajoutons à Hybrogen le support explicite de la C-SRAM par l'implémentation d'une passe d'analyse et de transformation. Cette passe va détecter des opérations dont les types sont associés à la mémoire C-SRAM. Les types utilisés sont les pointeurs vers des vecteurs stockés en mémoire. La nature paramétrable des types de données Hybrolang nous permet donc de couvrir l'ensemble des tailles de données supportées par l'architecture (Table 5.4). Le choix de l'utilisation de ces types de données fait que Hybrolang intègre un modèle de programmation de la mémoire C-SRAM explicitement vectorisé.

La Figure 5.11 présente la passe d'analyse et de transformation implémentée pour le support de la C-SRAM au sein de `Hybrolang`. La passe va utiliser les types de données pour détecter les variables C-SRAM et effectuer la macro-expansion des opérateurs en séries d'instructions HPU nécessaire pour la génération des instructions C-SRAM requises. La transformation statique d'opérations C-SRAM en séquences d'instructions HPU permet d'optimiser la génération des instructions C-SRAM au runtime.

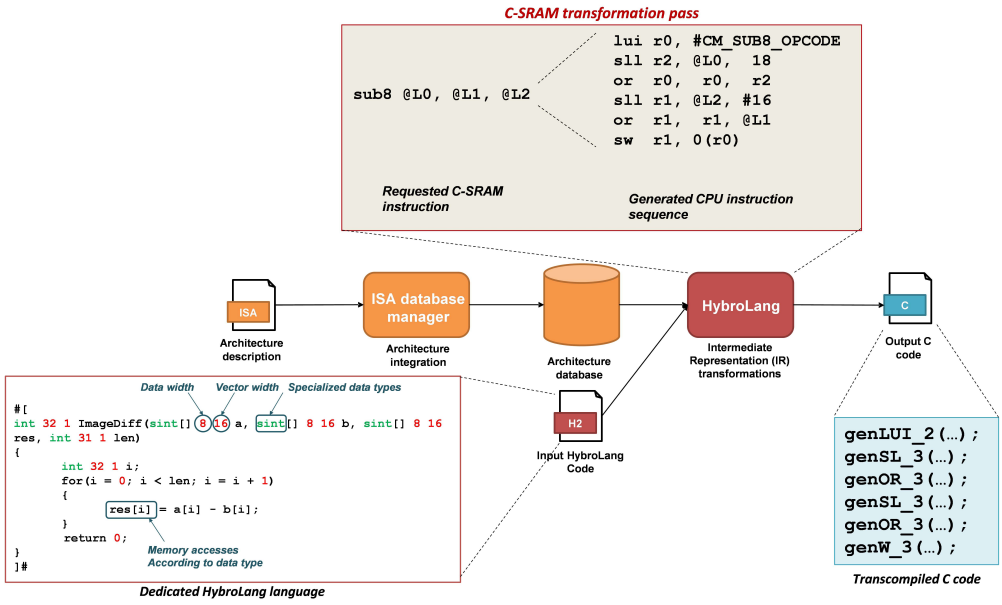


FIGURE 5.11 – La passe de transformation Hybrolang permet de supporter syntaxiquement les opérations C-SRAM pour un modèle de programmation bas niveau explicitement vectorisé.

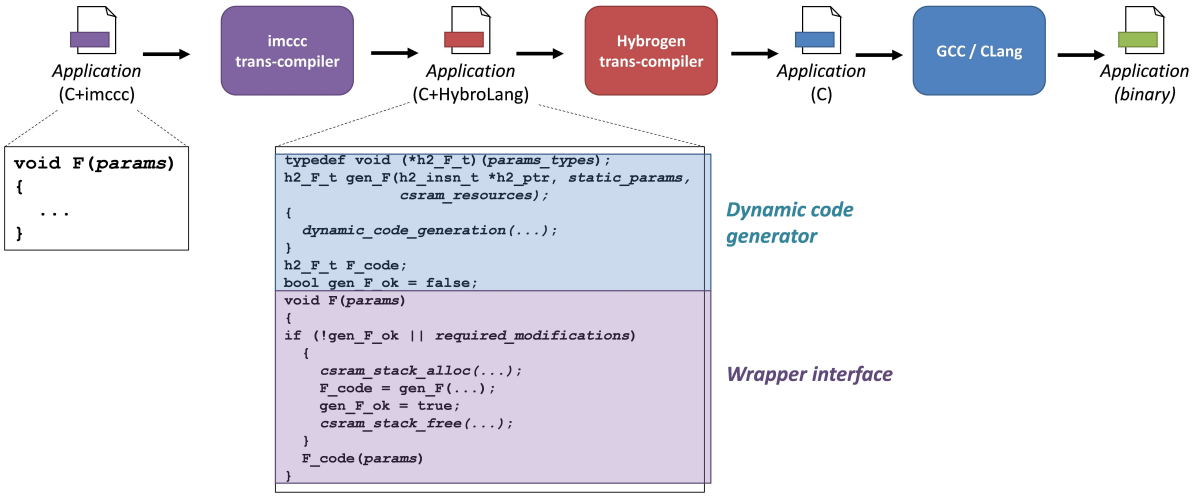


FIGURE 5.12 – Le langage Hybrolang peut être utilisé en interface d'interopération entre IMCCC et Hybrogen pour fournir la compilation dynamique à la C-SRAM.

Le langage Hybrolang ne décrit pas exactement le même modèle de programmation que le langage C. En effet, les codes écrits dans ce langage ne décrivent pas un code qui sera exécuté au runtime mais qui sera compilé dynamiquement au runtime. Pour concilier le modèle de programmation IMCCC avec Hybrolang, notre méthodologie effectue de l'analyse de code pour générer deux fonctions permettant d'implémenter la même fonctionnalité que le code original (Figure 5.12). La première est que nous appelons le *call-back* de la fonction, et qui correspond à la

fonction de génération de code contenant le code source transformé du IMCCC vers le Hybrolang. Ses paramètres sont le pointeur vers une région en mémoire à laquelle le code sera généré, ainsi que les éléments du code pouvant être rendues dynamiquement constantes. L'identification de paramètres de la fonction pouvant être transformées en constantes dynamiques requiert une analyse de code concernant ses invocations à l'échelle globale de l'application. La deuxième fonction est ce que nous appelons l'*interface* de la fonction. Cette interface possède les mêmes paramètres que la fonction `imccc originale` et compile – voire, recompile – la fonction si nécessaire grâce à l'appel à son call-back assigné.

```
void axpy(int A, int *X, int *Y, int len)
{
  imc_int a, x, y;
  int i;
  #pragma imc parallel for in
  for (i = 0; i < len; i += 1)
  {
    a = A;
    y = Y[i];
    y += a * x;
    Y[i] = y;
  }
}
```

FIGURE 5.13 – Une implémentation C-SRAM de la routine `axpy`, écrite en IMCCC.

La code présent à la Figure 5.13 présente une implémentation IMCCC de la routine `axpy`, qui effectue la multiplication par un scalaire d'un vecteur avant de l'accumuler.

La figure 5.14 présente la routine `axpy` une fois transformée vers du Hybrolang. La définition du call-back intègre bien le code qui sera dynamiquement compilé, avec les optimisations de code présentées dans la précédente section pour gérer les transferts de données (vectorisation du code, insertion d'un épilogue, etc.). La définition de l'interface contient les appels aux routines pour l'allocation statique en stockage C-SRAM, ainsi que la génération dynamique du code. Dans le présent exemple, les références vers variables C-SRAM ainsi que le paramètre `len` sont rendus statiques. Nous faisons la supposition que l'analyse de code faite pour la génération de l'interface renvoie que les invocations de la routine `axpy` ne fait pas varier ce paramètre, ce qui le permet d'être promu comme constante dynamique.

Conclusion

Nous avons présenté IMCCC, un modèle de programmation permettant la programmation de codes de stencils pour la mémoire C-SRAM et le contrôleur DMU. Ce modèle de programmation est implicitement vectorisé, grâce à des types de données dédiés ainsi que des directives, pour permettre une expression de voisinages de stencils de façon structurelle. Nous avons présenté une implémentation d'IMCCC, trans-compilant du code IMCCC vers du langage C. Nous avons également présenté une méthodologie utilisant l'environnement de compilation *Hybrogen* pour trans-compiler du code IMCCC vers du code de compilation dynamique.

```
// Dynamic axpy callback declatation;
(*h2_axpy_t)(int, int*, int*, int);
h2_axpt_t gen_axpy(h2_insn_t *h2_ptr, int *a, int *x, int *x, int len);
// Dynamic axpy callback definition
h2_axpy_t gen_axpy(h2_insn_t *h2_ptr, int *a_, int *x_, int *y_, int len)
{
#[
  void axpy(int 32 1 A, int[] 32 1 X, int[] 32 1 Y)
  {
    int[] 32 4 a = (int[] 32 4)#(a_);
    int[] 32 4 x = (int[] 32 4)#(x_);
    int[] 32 4 y = (int[] 32 4)#(y_);
    int 32 1 i;
    int 32 1 trim;
    a = A;
    for (i = 0; i+4 < #(len); i += 4)
    {
      y = Y[i, 4, single, 1, 1];
      y += a * x;
      Y[i, 4, single, 1, 1] = y;
    }
    if (i < len)
    {
      trim = len - i;
      y = Y[i, trim, single, 1, 1];
      y += a * x;
      Y[i, trim, single, 1, 1] = y;
    }
  }
  return;
]#
  return (h2_axpy_t)h2_ptr;
}
// Wrapper function for dynamic axpy callback code
bool gen_axpy_ok = false;
h2_axpy_t axpy_code;
void axpy(int A, int *X, int *Y)
{
  if (!gen_axpy_ok)
  {
    h2_insn_t h2_ptr = (h2_insn_t *)malloc(1024);
    int *a = __imc_stack_alloc(1);
    int *x = __imc_stack_alloc(1);
    int *y = __imc_stack_alloc(1);
    axpy_code = gen_axpy(h2_ptr, a, x, y);
    gen_axpy_ok = true;
    __imc_stack_free(3);
  }
  axpy_code(A, X, Y);
}
```

FIGURE 5.14 – La méthodologie d'interopération proposée transforme la routine présentée en Figure 5.13 en code Hybrolang, qui devient dynamiquement compilée à partir du même modèle de programmation.

Chapitre

6 Simulation d'architectures de calcul proche-mémoire couplées au dispositif de transfert

- *Comment mesurer de façon qualitative l'intégration système de la mémoire C-SRAM et du contrôleur DMU ?*
- *Comment mesurer de façon qualitative l'exécution d'applications C-SRAM/DMU ?*

Introduction

Au cours de la présente thèse, nous avons présenté les contributions suivantes :

— Le contrôleur DMU, pour le transfert efficace de données de stencil vers la mémoire C-SRAM.

— Le Pattern Stream Encoding, qui permet d'encoder dans le jeu d'instructions du contrôleur DMU des voisinages de stencils de façon compacte.

— Le modèle de programmation IMCCC, qui permet de programmer à haut niveau des applications utilisant des codes de stencil via la mémoire C-SRAM et le DMU.

— La méthodologie d'inter-opération entre Hybrogen et IMCCC pour la programmation d'applications C-SRAM qui supportent la compilation dynamique.

La question qu'il nous faut maintenant aborder est la suivante : *comment déterminer que l'exécution du logiciel sur le matériel est satisfaisante* ? Cette question implique dans un premier temps de pouvoir accéder à des données qui décrivent l'état de l'exécution de nos applications d'intérêt sur le matériel d'intérêt. Une difficulté, cependant, est l'accès à un dispositif qui permette d'effectuer cette exécution. Si la mémoire C-SRAM a précédemment été implémentée sur circuit intégré, pour mener des caractérisations physiques[NPG+20], le contrôleur DMU est un composant matériel qui ne dispose d'aucune implémentation physique – à l'heure où ces lignes sont écrites.

La réalisation d'un circuit-test ou même d'une description *Register-Transfer Level* (RTL) dans le seul cadre de la présente thèse n'est pas réalisable par contraintes de temps et de ressources. En effet, la vérification et la validation d'un circuit numérique requièrent une quantité de tests qui demandent beaucoup de temps et de savoir-faire, de la description RTL arbitraire jusqu'à une description pouvant être livrée auprès d'une fonderie. En opposition à la description matérielle, nous retrouvons des environnements de simulation à haut niveau, qui permettent d'obtenir un modèle comportemental dont les spécifications sont identiques à l'implémentation physique désirée.

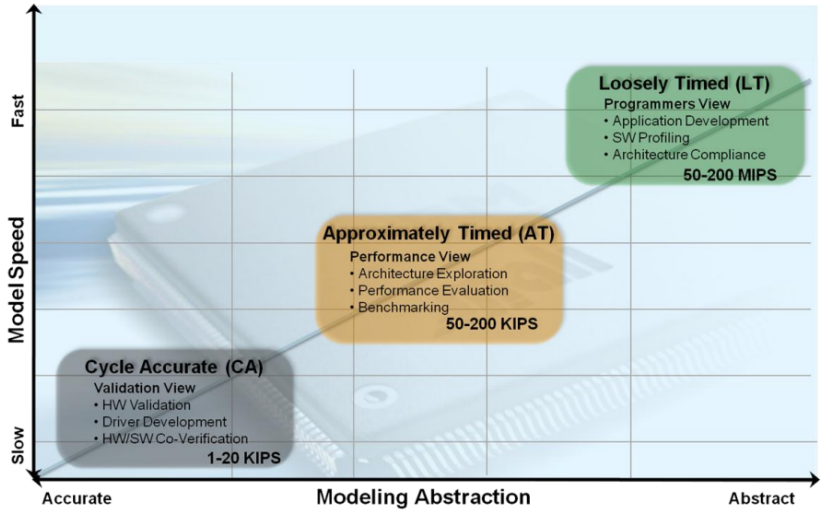


FIGURE 6.1 – Niveaux d'abstraction de simulation proposé par [NK12].

Il existe différents types de modèles de simulation, qui se distinguent par le degré d'abstraction employé. Le livre blanc publié par ARM[NK12] propose la classification présentée à la Figure 6.1, avec les paradigmes de simulation suivantes :

- *Loosely-Timed* (LT) : le modèle de simulation exhibe un comportement fonctionnel équivalent à une implémentation physique au niveau de ses entrées et de ses sorties, sans ne faire aucune garantie quant à son comportement micro-architectural. Il s'agit du plus haut degré d'abstraction en termes de simulation à temps discret, et il permet de vérifier des

fonctionnalités spécifiques des architectures de calcul au niveau logiciel pour une vitesse d'exécution très élevée.

- *Cycle-Accurate* (CA) : le modèle de simulation est strictement équivalent à une implémentation physique jusqu'à son comportement micro-architectural (à l'échelle de la porte logique, voire du transistor). Il s'agit du plus bas degré d'abstraction en termes de simulation à temps discret, et il permet d'effectuer des vérifications d'ordre matérielles dont les informations extraites sont pertinentes pour le développement de l'implémentation architecturale visée, au coût d'un temps de simulation beaucoup plus lent.
- *Approximately-timed* (AT) : le modèle de simulation adopte une approche hybride aux deux catégories précédemment citées, en utilisant un modèle d'abstraction intermédiaire (par exemple, à l'échelle de blocs logiques et de transactions entre ces derniers).

Chacun de ces paradigmes présente un rapport *temps d'exécution / précision du modèle* différent, mais qui est directement équivalent à un rapport *temps d'exécution / effort d'implémentation*. Il existe de nombreux environnements de simulation qui supportent plusieurs de ces paradigmes. Ainsi, la spécification SystemC permet la description en C++ d'architectures de calcul pour de la simulation CA[Pan01][Bha02][MRH+01] ainsi que de la simulation AT[GLD10][PHG11][GM]. Gem5 est un autre environnement de modélisation et de simulation d'architecture de calcul à la fonctionnalité proche du CA[BBB+11][BGOS12][PHO+14][LPAA+20]. Ces environnements de simulation intègrent des modèles de description complets et exhaustifs, mais présentent également un effort conséquent pour implémenter un modèle de simulation en avance de phase de la conception matérielle, pour l'exploration logicielle. De plus, ces environnements présentent des temps de simulation potentiellement très longs, de par leur niveau d'abstraction, pour l'exécution d'application dont la taille des jeux de données d'entrée peut s'élever à l'ordre du Giga-octet.

Pour affiner notre choix parmi une sélection de solutions de simulations présentées dans la littérature, nous dressons les observations suivantes :

- L'outil de simulation sur lequel nous baser doit nous permettre de modéliser l'intégration d'une mémoire C-SRAM ainsi que le contrôleur DMU au sein d'un système complet, plus précisément au sein d'une hiérarchie mémoire complète. Par souci de fidélité avec des architectures de calcul communes, nous souhaitons rendre possible la simulation de mémoires cache.
- L'outil de simulation doit nous permettre de lancer des applications prévues pour des systèmes complets, c'est-à-dire qui accèdent à des entrées/sorties tels que des disques ou d'autres types de périphériques. Nous jugeons que cette condition est essentielle pour permettre de lancer des applications avec des jeux de données d'entrée suffisamment grand pour stresser l'architecture. Notamment, nous attendons que les jeux de données d'entrée soient suffisamment grands pour pouvoir déborder de la capacité de l'ensemble de la hiérarchie mémoire, soit la capacité de stockage totale des mémoires caches et de la C-SRAM.
- L'outil de simulation doit nous permettre d'évaluer l'exécution des applications d'intérêt selon des métriques qualitatives liées à l'exécution de ces dernières. Cette contrainte implique la possibilité d'intégrer des compteurs de performances pour effectuer le profilage d'une application donnée sur un système de calcul donné.

Notre contribution, à la suite de l'établissement de ces contraintes, est une méthodologie de simulation pour l'évaluation d'applications utilisant la mémoire C-SRAM et le contrôleur DMU. Notre méthodologie de simulation se base sur pour la simulation de CPUs, et modélise par-dessus cette base un système mémoire contenant une hiérarchie mémoire associé à un modèle de performance pour effectuer une simulation AT dont des statistiques de performance peuvent être extraites. Cette méthodologie de simulation est ainsi employée pour évaluer l'impact de l'intégration du DMU au sein de la hiérarchie mémoire d'une architecture de référence, ainsi que l'impact de la compilation dynamique sur la performance des applications C-SRAM. Le chapitre se décompose comme suit : Section 6.1 présente notre flot de simulation basé sur QEMU. Section 6.2 présente notre méthodologie d'évaluation, sur la base de notre flot de simulation, ainsi que les résultats obtenus. Enfin, Section 6.3 conclut le chapitre.

6.1 Simulation de systèmes intégrant la C-SRAM et le DMU par le biais de QEMU

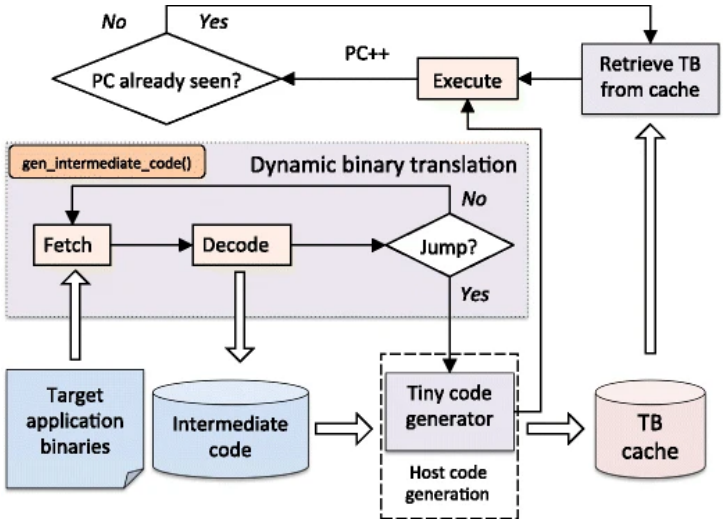


FIGURE 6.2 – Mécanisme de *Dynamic Binary Translation* au sein de l'émulateur QEMU. Image tirée de [FP16]

.

QEMU [Bel05] est principalement un outil de simulation de CPU se basant sur le principe de *Dynamic Binary Translation*. Ce mécanisme permet de traduire à la volée le code d'un binaire prévu pour une certaine architecture vers le jeu d'instructions de la machine hôte par le biais du *Tiny Code Generator*, une infrastructure de représentation intermédiaire. Le mécanisme de traduction dynamique, couplé à des stratégies de cache et de réutilisation de code traduit, permet d'obtenir des performances d'émulation du jeu d'instruction remarquables.

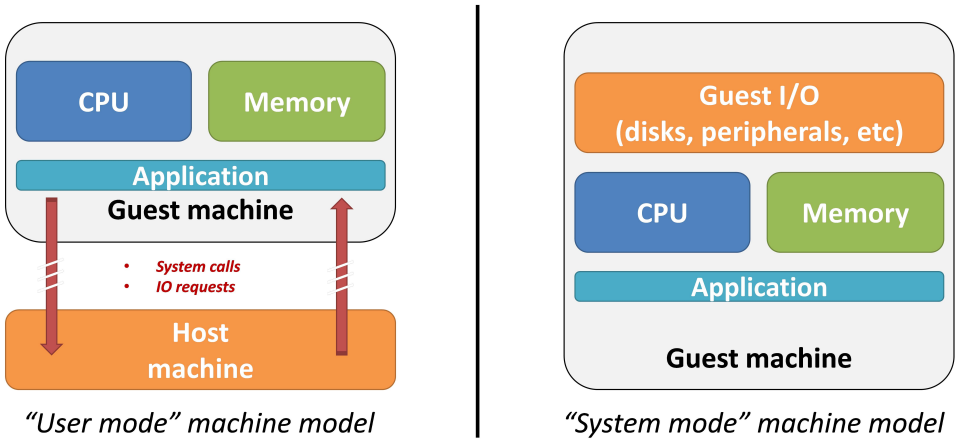


FIGURE 6.3 – En fonction du mode de fonctionnement de QEMU, différents types de machine peuvent être modélisés.

QEMU possède deux modes de fonctionnement. Le premier est nommé *User Mode*. Le binaire dédié au CPU invité est exécuté directement en tant que processus utilisateur du système d'exploitation de la machine hôte. Cette dernière s'engage à fournir au binaire invité le support de fonctionnalités spécifiques aux systèmes d'exploitation, tels que les appels systèmes ou l'accès à des périphériques. Le deuxième mode est appelé le *System mode*. Dans ce mode, QEMU modélise de façon complète un modèle de machine invitée, rattaché au CPU invité. Cette machine possède ses propres périphériques, disques et lignes d'interruption, et est donc capable de lancer un système d'exploitation de façon autonome. Nous remarquons que les deux modes de QEMU permettent de simuler des modèles de machine différents : le *User Mode* s'apparente à un système *semi-hosted* tandis que le *System Mode* émule une machine fonctionnant de façon autonome. Chacun de ces modèles de machine présente des avantages et des inconvénients : le *User Mode* permet l'exécution d'applications sur des machines émulées sans se soucier du support du matériel ou du système d'exploitation, tandis que le *System Mode* permet d'émuler l'exécution de systèmes d'exploitation.

Nous choisissons d'orienter l'implémentation de notre méthodologie vers le *User Mode* de QEMU pour plusieurs raisons. Premièrement, nous souhaitons focaliser nos efforts d'évaluation sur l'étude d'applications dans des domaines très spécifiques, et n'avons aucun intérêt à effectuer cette évaluation dans le cadre d'une exécution au sein d'un système d'exploitation. De plus, la gestion des interruptions, des entrées/sorties ainsi que des appels systèmes nous permet de focaliser nos efforts de modélisation sur l'architecture au plus proche du CPU, c'est-à-dire la hiérarchie mémoire. Ce point de focalisation est en accord avec notre objectif d'évaluer l'intégration de la mémoire C-SRAM et du contrôleur DMU au sein d'un système complet.

Cependant, le *User Mode* ne dispose pas nativement d'une API permettant la modélisation de tels composants, contrairement au *System Mode* qui dispose d'une API interne au projet de code QEMU. Il dispose néanmoins d'une API pour l'implémentation de *plug-ins*. Les plug-ins QEMU prennent la forme de librairies dynamiquement chargées lors de l'exécution de QEMU et qui se déclenchent selon des événements internes à QEMU, tels que lors de la traduction d'un

bloc de code.

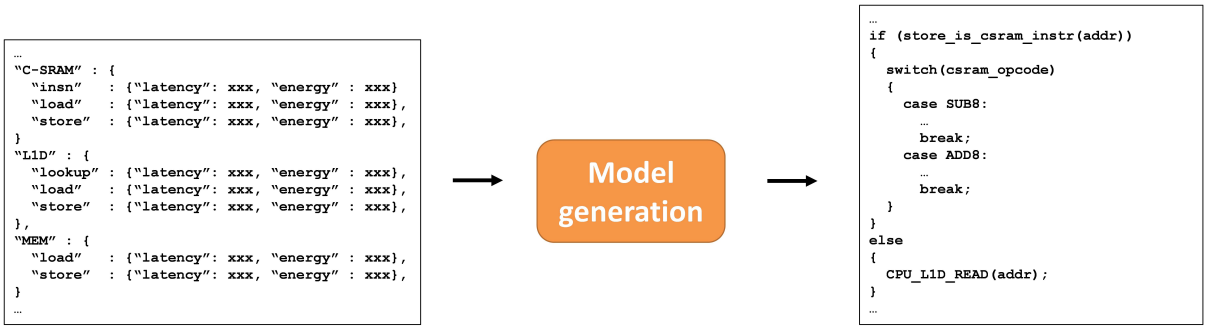


FIGURE 6.4 – L'API de plug-ins de QEMU peut être utilisé pour modéliser des comportements architecturaux et micro-architecturaux pour évaluation.

Nous implémentons la modélisation d'architecture sur la base de User-Mode QEMU en utilisant l'interface de plug-ins pour ajouter nos modifications. Notre méthodologie prend en entrée une description architecturale dans un format tel que le JSON. La description contient notamment une définition de la topologie de la hiérarchie mémoire, ainsi que les dimensions des composants. Un modèle de performance est également décrit pour les opérations internes de chaque composant (Figure 6.4). Ces chiffres de caractérisation sont récupérés depuis des sources telles que des articles, des data sheets, ou encore des environnements de modélisation de performance à l'échelle de la porte logique ou du transistor.

Component	Triggered events
CPU	Executed arithmetic/branch instruction, executed load instruction, executed store instruction
C-SRAM	Load request, Store request, Executed C-SRAM instruction
DMU	C-SRAM→C-SRAM transfer, C-SRAM→Mem. transfer, Mem.→C-SRAM transfer, On-wait
Cache memory	Load request, Store request, Cache look-up, Read miss, Write miss
Main memory	Load request, Store request

TABLE 6.1 – Liste des événements modélisés et ajoutés à QEMU pour la modélisation d'architectures C-SRAM.

L'intégration de ces nouveaux composants est modélisée par l'addition d'événements supplémentaires, qui sont décodés et déclenchés à partir de l'instruction CPU couramment exécutée (Table 6.1). Chaque événement possède un coût temporel et énergétique propre.

Le temps d'exécution total et le coût énergétique total d'une application exécutée et mesurée

par notre flot de modélisation peuvent être exprimés par les équations suivantes :

$$L_{total} = \sum_{I=First\ instruction}^{Up\ to\ completion} \mathcal{F}(I) \tag{6.1}$$

$$E_{total} = \sum_{I=First\ instruction}^{Up\ to\ completion} \mathcal{G}(I) \tag{6.2}$$

$\mathcal{F}$ et $\mathcal{G}$ sont les fonctions qui calculent respectivement le temps d'exécution et le coût de consommation dynamique de la hiérarchie mémoire globale à la suite de l'exécution de l'instruction, selon les paramètres des composants. Le DMU étant un composant non bloquant, la génération de certains de ses événements associés peut se superposer à ceux d'autres composants.

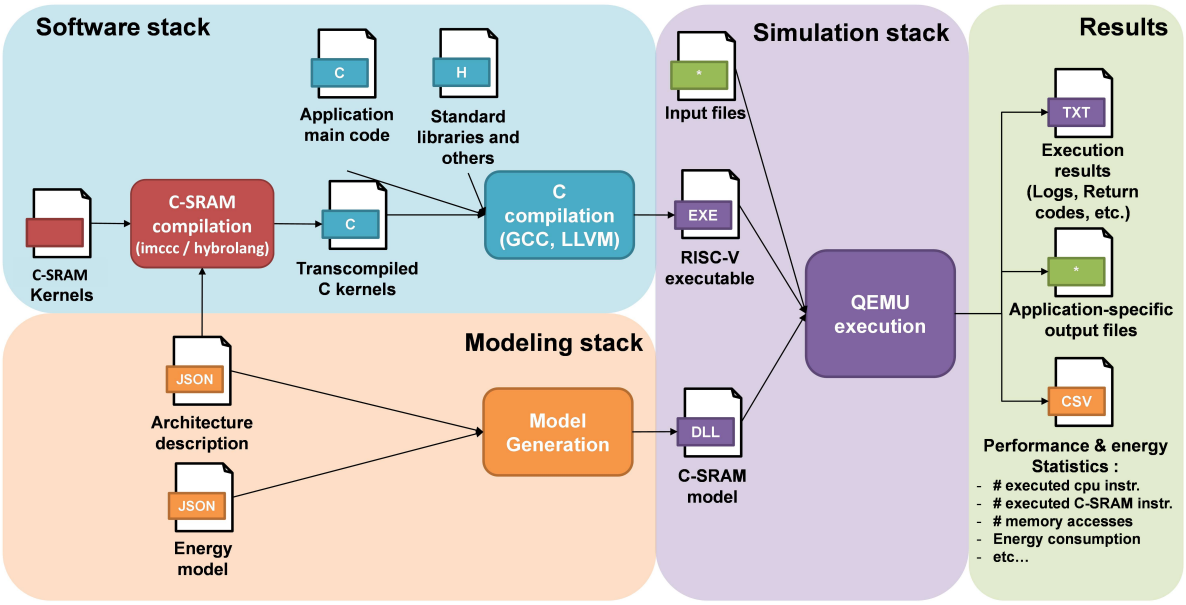


FIGURE 6.5 – La plate-forme expérimentale complète permet à partir d'une même description architecturale de générer le modèle de simulation QEMU ainsi que le support pour le compilateur.

La Figure 6.5 présente notre plate-forme expérimentale pour l'évaluation d'applications utilisant des architectures basées autour de la C-SRAM. Nous utilisons comme entrées principales un ensemble de fichiers JSON décrivant la composition de l'architecture expérimentale ainsi que son modèle de performance. La description architecturale sert d'entrée pour la génération du plug-in de modélisation architecturale pour *User-mode* QEMU ainsi que pour la compilation des applications C-SRAM. Ces applications sont écrites en IMCCC et peuvent être compilées de façon statique en langage C, ou de façon dynamique par l'intermédiaire de l'environnement Hybrogen.

6.2 Méthodologie expérimentale

Il nous est maintenant nécessaire, après la mise au point de la plate-forme d'expérimentation globale, d'établir une méthodologie expérimentale pour l'évaluation de nos contributions. Rappelons que ces contributions sont : le Data-locality Management Unit (DMU), un contrôleur mémoire pour le transfert de données de stencils, et *imccc*, un compilateur pour la mémoire C-SRAM et le DMU qui supporte la compilation statique ainsi que dynamique. Notre stratégie d'évaluation expérimentale consiste à répondre aux questions suivantes : "quels sont les gains de l'utilisation de la C-SRAM et du DMU apportés par le modèle de programmation proposé" ? Et, "quels sont les gains de performance apportés par la compilation dynamique, par le biais de Hybrogen" ?

Pour identifier des résultats significatifs dans l'élaboration des réponses à ces questions, nous déterminons trois paramètres méthodologiques généraux :

- **Les applications** sur lesquelles évaluer les contributions de la présente thèse. Des solutions telles que SPEC, PolyBench ou encore EmBench sont présentes dans la littérature. Nous notons cependant que ces solutions ne contiennent pas tous des codes de stencil, ou bien alors des codes correspondant à des codes de stencils mais qui utilisent de la transformation de données pour être conformes à des architectures généralistes. Pour cette raison, nous décidons de faire une évaluation sur un ensemble d'applications sélectionnées par nos soins. Cette sélection a été faite selon une étude de la littérature concernant les domaines du Traitement d'image, de la Vision par ordinateur ainsi que de l'algèbre linéaire.
- **Les architectures de calcul** sur lesquelles exécuter ces applications. L'intégration de la mémoire C-SRAM et du DMU se faisant au sein de la hiérarchie mémoire, nous déduisons un intérêt à ce que chaque architecture de calcul ait une hiérarchie mémoire différente pour en évaluer l'impact sur la performance relative de nos contributions. Le paramétrage de l'architecture de calcul exécutante permettra de fournir aux concepteurs matériels de notre équipe des informations intéressantes pour l'évolution de la mémoire C-SRAM.
- **Les métriques** selon lesquelles évaluer l'implémentation d'une application donnée sur une plate-forme donnée. Remarquons que nous avons fait émerger deux principales questions scientifiques auxquelles nous avons l'objectif de répondre par le biais de résultats expérimentaux, et chaque question appartient à un domaine particulier : logiciel concernant l'impact de la compilation dynamique, et matériel/logiciel concernant l'impact du contrôleur DMU et de sa programmation. Nous pouvons d'ores et déjà conclure que deux jeux de métriques seront nécessaires pour évaluer différents aspects des applications de façon qualitative.

6.2.1 Architectures de calcul

Nous décidons de baser nos architectures de calcul expérimentales autour de CPUs RISC-V. La spécification RISC-V présente l'intérêt, contrairement à d'autres architectures telles que Arm ou Intel x86, d'être open-source et de permettre à ses utilisateurs la conception d'extensions dans le jeu d'instructions. Nous souhaitons, dans le cadre de nos évaluations, paramétrer notre plate-forme expérimentale avec des caractérisations de CPUs RISC-V existantes dans la littérature. Nous établissons également une préférence pour des produits commerciaux, sur lesquels

des documentations techniques et des articles d'évaluation peuvent être trouvés au sein de la littérature.

Nous choisissons de focaliser notre recherche de caractérisations autour des cœurs de calcul développés par *SiFive*. Cette entreprise de conception de CPUs, dite *fabless* (sans usine de production), a beaucoup contribué à l'adoption industrielle de la spécification RISC-V. Par leur expérience, elle a donc conçu de nombreux cœurs de calcul qui sont non seulement documentés mais également, pour la plupart, incorporés dans des produits commerciaux.

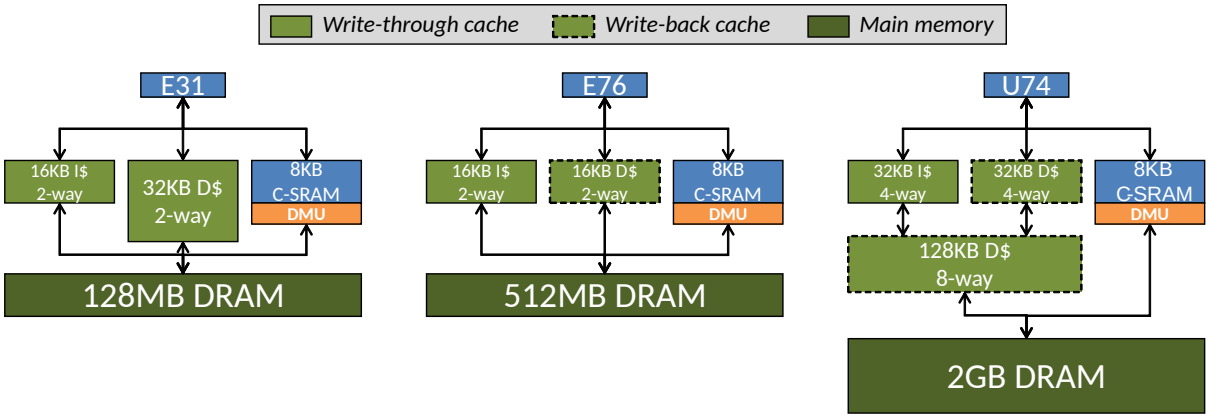


FIGURE 6.6 – Les trois architectures expérimentales employées pour l'évaluation d'applications C-SRAM, basées sur des CPUs SiFive.

La Figure 6.6 présente les trois architectures de calcul que nous avons sélectionnées pour nos évaluations expérimentales. L'intérêt de ces architectures est que chacune possède, du nœud de calcul le moins au plus rapide, une progression graduelle de la complexité de la hiérarchie mémoire. Les cas d'usages de telles architectures vont des solutions embarquées à faible consommation vers des nœuds de calcul haute-performance.

La première est un système cadencé à 320MHz et se base sur un SiFive E31[SiFb], avec un cache d'instructions et un cache de données tous deux de 16 Kilo-octets avec deux voies d'associativité. Les deux caches possèdent une politique d'écriture dite *write-through*. La seconde est un système cadencé à 800Mhz et se base sur un SiFive E76[Whe19], et possède les mêmes dimensions de hiérarchie mémoire que la précédente architecture, à la différence que le cache de données implémente une politique d'écriture *write-back*. Enfin la troisième architecture, cadencée à 1GHz, se base sur le SiFive U74[SiFa] et intègre une hiérarchie de cache à deux niveaux. Les caches L1 sont de capacité 32 Kilo-octets avec 4 voies d'associativité, tandis que le cache L2 est de 128 Kilo-octets avec 8 niveaux d'associativité. Tous les caches de la hiérarchie mémoire possèdent une politique d'écriture write-back.

Pour la modélisation de la C-SRAM, nous choisissons l'unité présentée dans le papier [NPG+20]. Cette unité C-SRAM est gravée en technologie 22nm et peut fonctionner jusqu'à une fréquence d'1 GHz. Elle dispose d'une capacité totale de 8 Kilo-octets et une taille de vecteurs de 128 bits. Le contrôleur DMU, quand a lui, est modélisé avec un port de lecture/écriture 128-bit vers la C-SRAM et un second port de lecture/écriture 32-bit vers la mémoire principale.

	E31-based *384MHz*		E76-based *800MHz*		U74-based *1GHz*	
	Latency (cycles)	Energy Cost	Latency (cycles)	Energy Cost	Latency (cycles)	Energy Cost
8KB C-SRAM	1	31.74pJ	2	31.74pJ	3	31.74pJ
L1 I$	1	R : 19pJ W : 25pJ	1	R : 19pJ W : 25pJ	1	R : 24pJ W : 24pJ
L1 D$	1	R : 34pJ W : 34pJ	1	R : 34pJ W : 34pJ	1	R : 24pJ W : 24pJ
L2 $					12	R : 52pJ W : 52pJ
DRAM	7	R : 8.17nJ W : 8.04nJ	24	R : 14.45nJ W : 14.35nJ	48	R : 39nJ W : 37.5nJ

TABLE 6.2 – Les paramètres de caractérisations des architectures expérimentales, obtenus avec CACTI 6.0 dans une technologie de gravure de 22nm.

Nous utilisons CACTI[MBJ09], un modèle de technologies mémoire, pour calculer en fonction des fréquences de fonctionnements globaux de chaque nœud de calcul, une approximation des latences et consommations énergétiques observées sur le circuit potentiellement fondu. La Table 6.2 présente les caractéristiques obtenus pour chaque architecture. Ces caractéristiques sont obtenues en paramétrant CACTI pour utiliser une technologie 22nm, afin qu'elles soient cohérentes avec les caractérisations de la mémoire C-SRAM précédemment mentionnées. Chaque architecture possède une mémoire principale DRAM de taille différente, dont la taille varie de 128 Méga-octets pour le nœud de calcul E31 jusqu'à 2 Giga-octets pour le U74.

6.2.2 Applications

Nous basons notre sélection d'applications selon les critères suivants : les applications sélectionnées doivent présenter des comportements suffisamment variés lors de l'exécution pour offrir une couverture de cas d'étude satisfaisante, et elles doivent être trouvées dans la littérature concernant nos domaines applicatifs d'intérêt. Nous rappelons que les codes de stencil sont une catégorie de solutions numériques particulièrement appréciée ans les domaines du traitement d'image, de la vision par ordinateur ainsi que de l'algèbre linéaire. Nous retenons donc ces domaines applicatifs comme d'intérêt.

Application	**Vector element size**	**Complexity**	**Pattern type**	**Avgerage data redundancy**
Frame differencing [Sin14]	8-bit	$\mathcal{O}(n)$	On-load : row-major On-store : row-major	1
Laplace filter [Opea]	16-bit	$\mathcal{O}(n)$	On-load : stencil (5-pt) On-store : row-major	≈ 5
Sobel filter [Opeb]	16-bit	$\mathcal{O}(n)$	On-load : stencil (9-pt) On-store : row-major	≈ 18
Matrix multiplication (squares) [Sta01]	32-bit	$\mathcal{O}(n^3)$	On-load : row/col-major On-store : row-major	n^2
Demosaicking (AoS / SoA) [MHC04]	16-bit	$\mathcal{O}(n)$	On-load : stencil (13-pt) On-store : irr.→row-major	≈ 26

TABLE 6.3 – Liste des applications retenues pour l'évaluation expérimentale, accompagnées de leurs attributs.

La Table 6.3 présente notre sélection d'applications après considération de nos critères d'intérêt. Chacune de ces applications correspond à une routine couramment utilisée dans son domaine associé, et certaines font partie d'interfaces logicielles standards telles que les APIs OpenVX[Gro17] ou encore OpenCV[Bra00] :

La différence d'image (*frame differencing*) effectue une différence saturée à zéro une image de référence correspondant aux éléments statiques d'une scène capturée, et les images consécutives d'un flux vidéo afin de détecter les éléments différents au sein de la scène. Cette solution a été historiquement employée pour la vision par ordinateur, car elle permet l'implémentation de détection de mouvement rudimentaire, à un faible coût arithmétique. Nous implémentons une version simplifiée de la différence d'image, qui effectue la détection de mouvement image par image seulement.

$$Out = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * I \tag{6.3}$$

$$Out = |G_x| + |G_y|, \begin{cases} G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \\ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I \end{cases} \tag{6.4}$$

Nous implémentons également deux filtres couramment employés en traitement d'image pour effectuer de la détection de contours : le *filtre de Sobel* et l'opérateur de Laplace. Le premier est implémenté sous la forme d'un filtre de convolution unique (Equation 6.3) tandis que le second utilise deux filtres pour calculer les composantes horizontales et verticales de l'image avant de les additionner (Equation 6.4). Chacun de ses deux filtres peut être utilisé en fonction des attributs de l'image d'entrée.

Le dématriçage (*demosaicking*) est une opération employée sur les capteurs d'image dits *à matrice de Bayer*. Une matrice de Bayer est une matrice de filtres verts, bleus et rouge, superposée au capteur d'image afin que chaque cellule de sa matrice de photo capteurs ne puisse recevoir qu'une couleur. Il s'agit de la technologie de séparation de couleur la plus couramment employée, pour sa simplicité de conception et ainsi son coût moins onéreux en comparaison d'autres technologies[HAMR22].

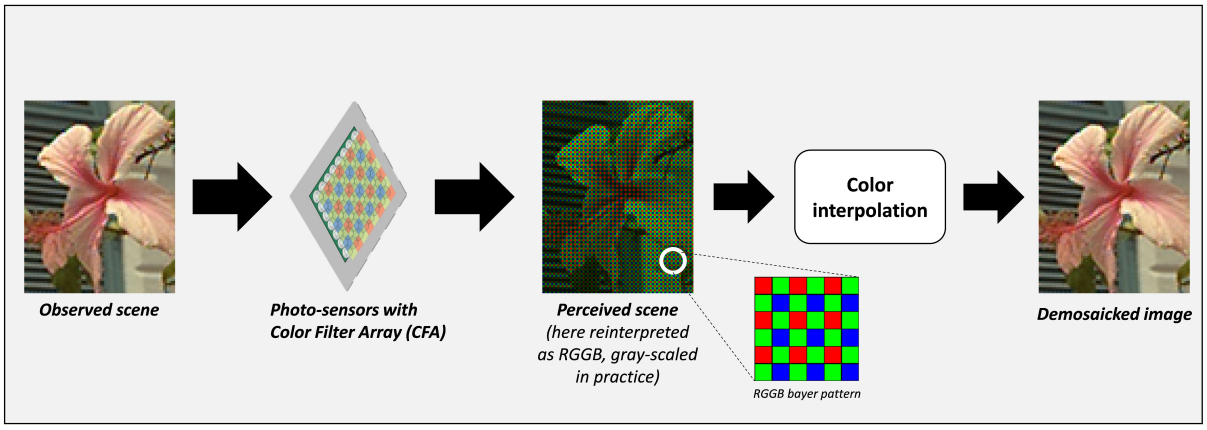


FIGURE 6.7 – La scène photographiée par le capteur d'image de Bayer est capturée sous d'image d'intensités lumineuses avant d'être recolorisée.

Pour obtenir une image numérisée en couleur, dans un format tel que le RGB, il est donc nécessaire d'employer des opérateurs mathématiques d'interpolation, qui permettent de calculer les composantes manquantes de chaque pixel de l'image acquise en fonction du reste de l'image (Figure 6.7). Cette opération d'interpolation est l'opération communément appelée *dématriçage*.

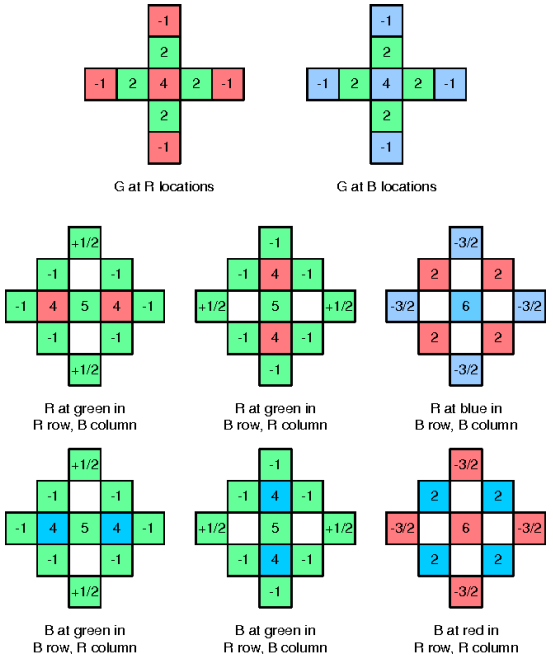


FIGURE 6.8 – L'opérateur d'interpolation bilinéaire discret, implémenté en un jeu de stencils pour un coût arithmétique moins élevé qu'une implémentation non-discrète à virgule flottante.

De nombreuses implémentations du dématriçage existent dans la littérature. Nous avons choisi d'implémenter l'algorithme de Malvar-He-Cutler[MHC04] pour évaluation. Cet algorithme

discrétise l'opérateur d'interpolation bilinéaire, normalement calculé en virgule flottante, sous la forme d'un code de stencil utilisant 8 filtres répartis en quatre voisinages (Figure 6.8). Cette implémentation du dématriçage présente l'avantage d'être compétitive en termes de qualité de colorisation numérique avec d'autres algorithmes de la littérature, tout en requérant une charge arithmétique moins coûteuse[Get11].

Enfin, la multiplication matricielle est une opération arithmétique très couramment utilisée dans de nombreux domaines, et elle fait partie de la spécification BLAS sous la forme de la GEneral Matrix Multiplication (`gemm`). Nous implémentons une version effectuant du partitionnement de boucle, afin de maximiser la réutilisation de données au sein de la C-SRAM.

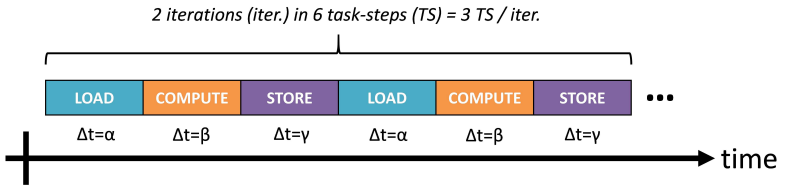
Rappelons que la performance d'une application donnée sur une architecture donnée est intrinsèquement corrélée à son implémentation. Nous voyons dans cette observation un intérêt d'étudier également l'impact de l'implémentation d'applications par le biais de nos contributions sur la performance des applications implémentées.

6.2.3 Optimisations du code

Dans cette sous-section, nous présentons des optimisations de code appliquées à haut niveau pour améliorer la performance des implémentations logicielles développées avec IMCCC. Dans le cadre de notre procédé expérimental, nous implémentons lors que possible plusieurs versions de nos applications expérimentales, qui diffèrent par l'utilisation ou non de ces techniques lorsque possible.

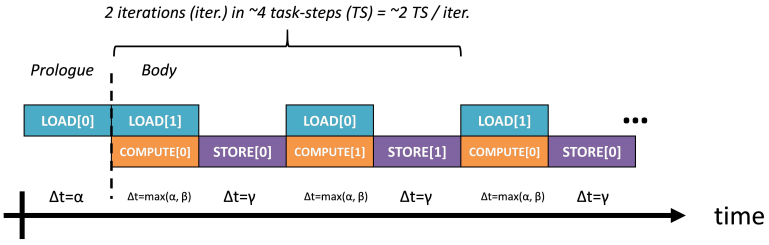
Une première optimisation de code que nous employons est le *double-buffering*. Cette optimisation consiste à recouvrir les périodes d'accès mémoire par des calculs afin d'optimiser l'utilisation de la bande passante.

```
int i = 0;
int tmpA, tmpB, tmpC;
for(i = 0; i < N; i += 1)
{
  // Blocking input transfers
  // due to data dependency
  tmpA = A[i];
  tmpB = B[i];
  // Computation
  tmpC = tmpA - tmpB;
  // Blocking output transfers
  // due to data dependency
  C[i] = tmpC;
}
```



(a) Temporisation en série.

```
int i = 0;
int idx = 0, nidx = 1;
int tmpA[2], tmpB[2], tmpC[2];
// Prologue
tmpA[idx] = A[i];
tmpB[idx] = B[i];
// Main body
for (; i < N; i += 1)
{
  if (i < (N-1))
  {
    // Data-independent input transfers
    tmpA[nidx] = A[i];
    tmpB[nidx] = B[i];
  }
  // Computation
  tmpC[idx] = tmpA[idx] - tmpB[idx];
  // Data-independent output transfers
  C[i-1] = tmpC[idx];
  // Update of buffer indices
  swap(bufno, bufno2);
}
```



(b) Temporisation en *double-buffering*.

FIGURE 6.9 – Chronogrammes d'une application en fonction des stratégies logicielles de temporisation employées.

Pour montrer l'impact du double-buffering sur un code, nous présentons aux Figures 6.9a et 6.9b l'exemple de codes qui effectue la soustraction de deux vecteurs A et B avant de stocker le résultat dans C. Ces codes peuvent être interprétés comme des ensembles des tâches de lecture (`LOAD`), des tâches de calcul (`COMPUTE`) et des tâches d'écriture (`STORE`), respectivement de temps d'exécution α, β et γ au sein du présent code. Les variables *tmpA*, *tmpB* et *tmpC* sont les tampons stockant les données respectivement en relation avec A, B et C. Ces tâches sont situées au sein d'une boucle, dont la variable d'induction est `i`.

Le code de la Figure 6.9a utilise une stratégie de temporisation employée dite *en série*. Les variables `tmpA`, `tmpB` et `tmpC` sont toutes trois des variables scalaires. Nous appelons respectivement `LOAD`, `COMPUTE` et `STORE` l'ensemble des lectures depuis la mémoire principale vers `[tmpA,tmpB]`, le calcul de `tmpC` et l'écriture depuis `tmpC` vers la mémoire principale. Nous pouvons observer une dépendance de donnée immédiate d'une tache à l'autre, qui ne peut pas être résolue au niveau sémantique. En conséquence, la durée d'exécution moyenne d'une itération de la boucle au sein du code sera égale à $\alpha + \beta + \gamma$.

Le code de la Figure 6.9b optimise le code de la Figure 6.9a via l'utilisation du double-buffering. Les variables `tmpA`, `tmpB` et `tmpC` sont modifiées pour ne plus être des scalaires mais des tableaux de deux éléments. Nous définissons `LOAD[idx]`, `COMPUTE[idx]` et `STORE[idx]` comme les taches manipulant les données stockées à `tmpA[idx]`, `tmpB[idx]` et `tmpC[idx]`. Le code effectue avant l'entrée de la boucle un premier chargement des données nécessaires lors

de la première entrée dans la boucle – c'est-a-dire, pour `i=0`, depuis la mémoire principale vers `[tmpA[0],tmpB[0]]`. Lors de cette première entrée dans la boucle, `LOAD[1]` transfert de façon anticipée les données nécessaires pour l'itération `i=i+1`, puis exécute `COMPUTE[0]` qui calcule le résultat à l'itération courante `i`. Ce résultat est ensuite écrit vers la mémoire principale par `store[0]`. Les index des variables sont ensuite commutés pour l'itération suivante. En partant du principe que les accès mémoire de lecture peuvent être faits de façon non bloquante, il est possible de recouvrir la tâche `LOAD[1]` sur `COMPUTE[0]`. La durée d'exécution moyenne d'une itération de boucle sera donc égale à $\gamma + max(\alpha, \beta)$.

Il est possible d'utiliser le double-buffering sur les architectures qui intègrent la C-SRAM et le DMU, car ce dernier est capable d'effectuer des transferts de données non bloquants. Nous implémentons donc dans notre compilateur des options pour sélectionner et paramétrer, parmi les accès mémoires d'un code IMCCC à compiler, les accès mémoires à effectuer de façon non bloquante. Cependant, car nous avons modélisé dans le cadre de la présente thèse un DMU qui ne dispose que d'un seul port d'entrée et de sortie, nous ne pouvons pas implémenter des applications dont le temps d'exécution est optimal – c'est-à-dire égal à $max(\alpha, \beta, \gamma)$.

L'utilisation du double-buffering lors de la vectorisation de boules nous permet d'implémenter des *pipelines logiciels* à haut niveau. Les transformations de code nécessaires pour implémenter ces pipelines logiciels sont identifiables et pourraient donner lieu, à l'avenir, à un flot d'automatisation de pipelining logiciel pour les applications développées en IMCCC.

6.3 Résultats expérimentaux

6.3.1 Impact de la programmation du contrôleur DMU sur la performance applicative

Par le biais d'IMCCC, nous rédigeons à haut niveau deux versions de code pour chaque application : une qui implémente une temporisation de données en série et une autre qui effectue du pipeline logiciel. Pour la suite de cette section et de la section suivante, l'abbreviation **SWP** (pour *Software Pipeline*) décrira le pipeline logiciel. Chaque version de code – avec et sans SWP – est compilée pour le HPU pour exécution scalaire, et pour la C-SRAM en quatre autres versions.

Version	Canal de lecture DRAM	Canal d'écriture DRAM
DD	DMU	DMU
DC	DMU	Hiérarchie de cache
CD	Hiérarchie de cache	DMU
CC	Hiérarchie de cache	Hiérarchie de cache

TABLE 6.4 – versions possibles pour un code à une entrée et une sortie.

Ces versions se distinguent par l'association des lectures et écritures sur les différents canaux de transferts de données disponibles sur l'architecture ciblée. Ainsi, chaque application possédant un flux d'entrée et un flux de sortie, l'association de chaque sur le canal D ou le canal C génère

au total 4 versions (voir Table 6.4). Ce sont donc, pour chaque application, 9 versions qui sont compilées, exécutées et évaluées pour différents jeux de données d'entrée.

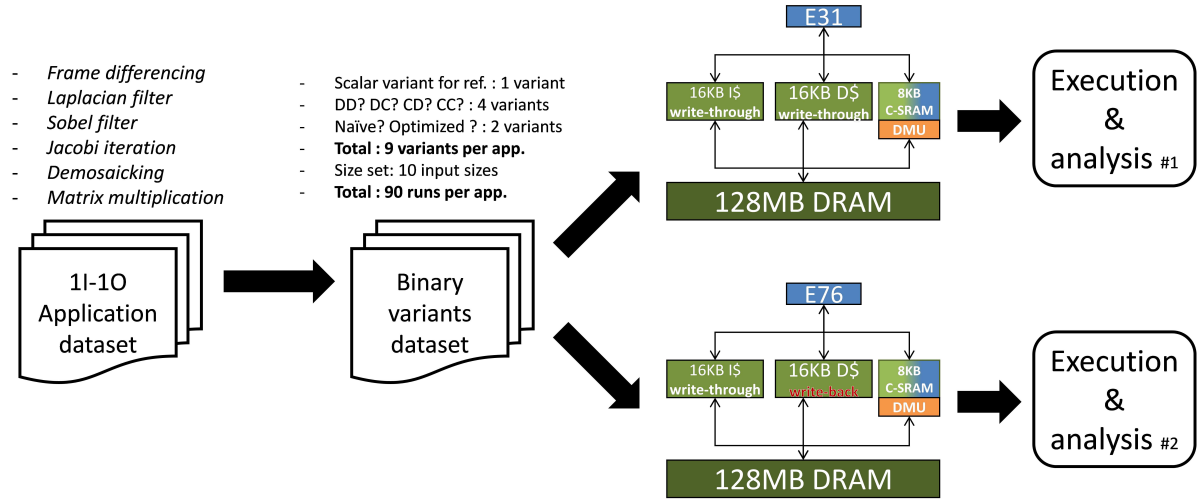
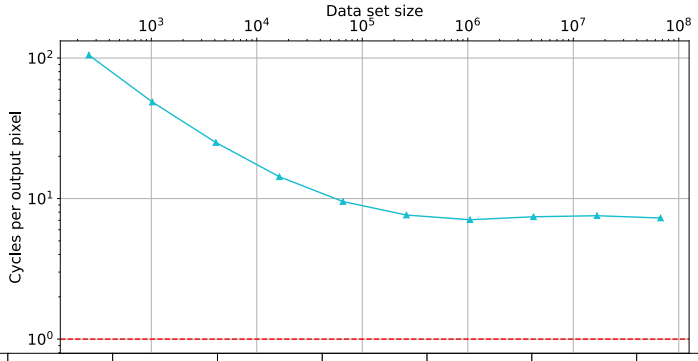


FIGURE 6.10 – Pour chaque application C-SRAM, une implémentation donnée donne lieu à plusieurs versions qui utilisent différemment la bande passante de l'architecture.

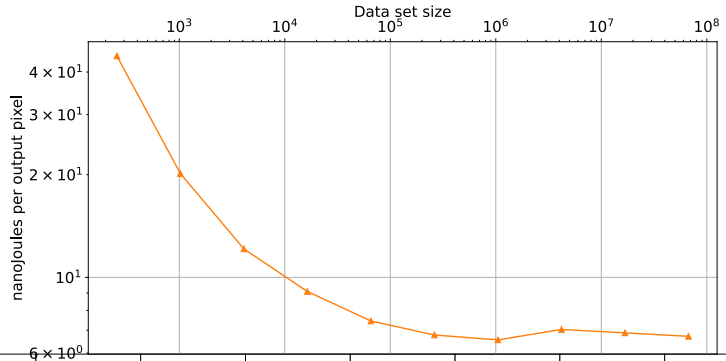
Nous exécutons ces différentes versions sur deux architectures différentes, celles basées sur les cœurs E31 et E76 (Figure 6.10). Ces deux architectures présentent des performances intrinsèques différentes mais surtout, des politiques de réécriture différentes au niveau de leur cache de données, ce qui va nous permettre d'évaluer l'intégration du contrôleur DMU selon la hiérarchie mémoire du HPU.

À titre d'exemple, nous présenterons les résultats exhaustifs de l'exécution d'une application – la différence d'image – sur les architectures E31 et E76.



	risc-v, naive	DD, naive	DC, naive	CD, naive	CC, naive	DD, db	DC, db	CD, db	CC, db
256 B	132.504	104.926	232.668	277.168	402.914	109.754	223.539	318.738	442.605
1 KB	130.001	49.247	178.87	132.308	261.307	48.639	170.697	151.541	280.026
4 KB	140.688	25.304	155.272	69.397	199.522	25.04	150.088	78.305	209.288
16 KB	147.289	15.244	144.767	39.021	168.603	14.3	141.01	43.538	173.189
64 KB	148.076	10.334	139.163	22.427	152.295	9.529	136.27	24.655	154.571
256 KB	148.131	8.068	136.495	14.334	144.257	7.635	134.105	15.444	145.401
1 MB	333.745	7.07	135.322	10.398	140.383	7.707	134.075	15.766	145.489
4 MB	674.148	7.423	135.902	11.298	140.799	7.831	134.052	13.8	143.523
16 MB	674.137	7.551	135.847	10.985	140.764	7.831	133.978	13.776	143.438
64 MB	674.131	7.273	135.57	10.255	140.163	7.614	133.783	13.118	142.78

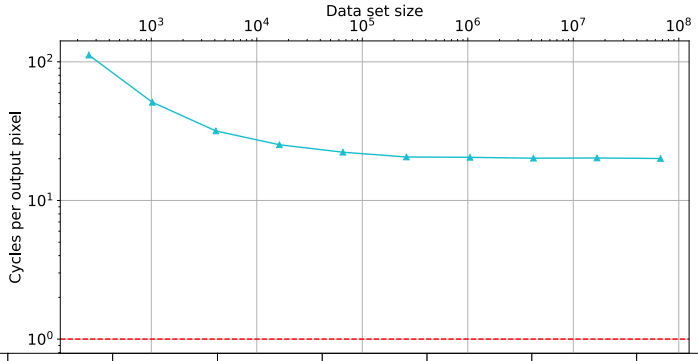
(a) Performance temporelle, en Cycles Par Point.



	risc-v, naive	DD, naive	DC, naive	CD, naive	CC, naive	DD, db	DC, db	CD, db	CC, db
256 B	1.32e+02	4.46e+01	1.70e+02	2.25e+02	3.48e+02	7.61e+01	1.88e+02	2.87e+02	4.08e+02
1 KB	1.30e+02	2.02e+01	1.47e+02	1.06e+02	2.32e+02	3.39e+01	1.53e+02	1.35e+02	2.61e+02
4 KB	1.41e+02	1.21e+01	1.39e+02	5.64e+01	1.83e+02	1.89e+01	1.42e+02	7.04e+01	1.98e+02
16 KB	1.47e+02	9.09e+00	1.36e+02	3.26e+01	1.59e+02	1.23e+01	1.37e+02	3.97e+01	1.66e+02
64 KB	1.48e+02	7.45e+00	1.34e+02	1.93e+01	1.46e+02	8.98e+00	1.35e+02	2.28e+01	1.49e+02
256 KB	1.48e+02	6.78e+00	1.34e+02	1.28e+01	1.39e+02	7.52e+00	1.34e+02	1.46e+01	1.41e+02
1 MB	3.39e+02	6.56e+00	1.33e+02	9.72e+00	1.36e+02	7.49e+00	1.34e+02	1.49e+01	1.41e+02
4 MB	6.88e+02	7.04e+00	1.34e+02	1.06e+01	1.37e+02	7.72e+00	1.34e+02	1.32e+01	1.40e+02
16 MB	6.88e+02	6.87e+00	1.34e+02	1.03e+01	1.37e+02	7.66e+00	1.34e+02	1.32e+01	1.39e+02
64 MB	6.88e+02	6.72e+00	1.34e+02	9.69e+00	1.36e+02	7.50e+00	1.34e+02	1.26e+01	1.39e+02

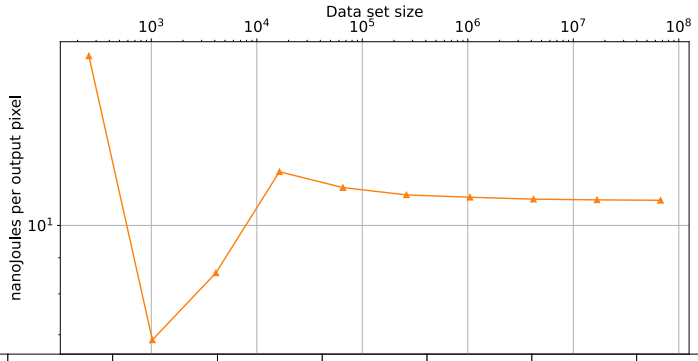
(b) Performance énergétique, en nano-Joules Par Point.

FIGURE 6.11 – Profil d'exécution de la différence d'image sur l'architecture basée sur le E31.



	risc-v, naive	DD, naive	DC, naive	CD, naive	CC, naive	DD, db	DC, db	CD, db	CC, db
256 B	12.27	166.957	156.699	148.23	145.383	130.293	118.43	115.133	111.805
1 KB	11.083	89.099	85.144	70.006	74.018	62.716	57.344	51.073	53.717
4 KB	21.431	53.099	56.02	42.176	53.704	35.882	38.289	31.732	43.945
16 KB	39.643	36.814	46.629	39.387	55.738	25.222	34.564	34.018	50.298
64 KB	34.601	28.014	37.443	28.509	43.794	22.288	28.303	26.312	42.215
256 KB	36.653	23.499	32.692	23.54	39.175	20.593	25.114	22.273	38.075
1 MB	58.642	21.397	30.645	21.29	36.968	20.474	26.039	22.983	40.144
4 MB	58.385	21.151	30.348	21.131	37.094	20.18	24.98	21.881	38.563
16 MB	58.16	21.04	30.219	22.508	36.907	20.252	25.024	21.828	38.492
64 MB	58.072	20.781	29.95	21.398	36.617	20.077	24.752	21.572	38.102

(a) Performance temporelle, en Cycles Par Point.



	risc-v, naive	DD, naive	DC, naive	CD, naive	CC, naive	DD, db	DC, db	CD, db	CC, db
256 B	3.86e+00	2.71e+01	2.26e+01	2.27e+01	1.74e+01	3.28e+01	2.84e+01	2.67e+01	2.13e+01
1 KB	3.17e+00	1.56e+01	1.36e+01	9.13e+00	6.96e+00	1.66e+01	1.40e+01	9.74e+00	6.88e+00
4 KB	9.14e+00	1.27e+01	1.39e+01	8.82e+00	1.06e+01	1.26e+01	1.36e+01	8.57e+00	1.08e+01
16 KB	1.97e+01	1.19e+01	1.66e+01	1.51e+01	1.95e+01	1.19e+01	1.65e+01	1.49e+01	1.92e+01
64 KB	1.68e+01	1.13e+01	1.56e+01	1.28e+01	1.64e+01	1.13e+01	1.56e+01	1.29e+01	1.70e+01
256 KB	1.80e+01	1.11e+01	1.50e+01	1.19e+01	1.57e+01	1.10e+01	1.51e+01	1.19e+01	1.58e+01
1 MB	3.07e+01	1.10e+01	1.49e+01	1.16e+01	1.54e+01	1.10e+01	1.57e+01	1.23e+01	1.70e+01
4 MB	3.05e+01	1.09e+01	1.48e+01	1.15e+01	1.54e+01	1.10e+01	1.54e+01	1.20e+01	1.64e+01
16 MB	3.04e+01	1.09e+01	1.47e+01	1.14e+01	1.53e+01	1.10e+01	1.54e+01	1.19e+01	1.63e+01
64 MB	3.03e+01	1.09e+01	1.47e+01	1.14e+01	1.53e+01	1.09e+01	1.53e+01	1.19e+01	1.62e+01

(b) Performance énergétique, en nano-Joules Par Point.

FIGURE 6.12 – Profil d'exécution de la différence d'image sur l'architecture basée sur le E76.

Les Figures 6.11 et 6.12 présentent les profils d'exécution de la différence d'image sur les architectures basées sur le SiFive E31 et sur le SiFive E76. Chaque figure se décompose en deux sous-figures : la première présente la performance temporelle, en Cycles Par Point, et la seconde présente la performance énergétique en nano-Joules Par Point. Nous choisissons ces métriques *de metier* car elles permettent de comparer nos résultats avec ceux avec l'art antérieur de façon neutre, sans tenir compte de subtilités micro-architecturales ou liées aux technologies de conception et de gravure de chaque solution comparée.

Pour chacune de ces métriques, nous traçons une courbe dont l'abscisse est la taille des structures de données traitées, et l'ordonnée la meilleure performance obtenue parmi toutes les versions logicielles de la différence d'image. Les tableaux présents dans chaque sous-figure permettent d'identifier la version avec la meilleure performance pour chaque taille de jeux de données, et pour chaque métrique d'évaluation.

Sur les deux architectures, nous pouvons observer que l'utilisation de la C-SRAM pour la différence d'image devient pertinente à partir de jeux de données supérieurs 4 Kilo-octets. Une fois ce seuil passé, nous observons des résultats intéressants quant à l'utilisation du SWP sur chaque architecture.

Taille d'image	**16 KB**	**64 KB**	**256 KB**	**1 MB**	**4 MB**	**16 MB**	**64 MB**
Sans SWP	39.642	34.601	36.653	58.642	58.835	58.160	58.072
Avec SWP	14.300 -36.37%	9.529 -35.76%	7.635 -43.81%	7.707 -61.67%	7.831 -65.70%	7.831 -65.18%	7.614 -65.43%

(a) Performance temporelle (Cycles Par Point). Le plus bas est le meilleur.

Image size	**16 KB**	**64 KB**	**256 KB**	**1 MB**	**4 MB**	**16 MB**	**64 MB**
Sans SWP	1.19	1.13	1.11	1.10	1.09	1.09	1.09
Avec SWP	1.19 +0.00%	1.13 +0.00%	1.11 +0.00%	1.10 +0.00%	1.09 +0.00%	1.09 +0.00%	1.09 +0.00%

(b) Performance énergétique (nano-Joules Par Point). Le plus bas est le meilleur.

TABLE 6.5 – L'utilisation du Software Pipelining (SWP) sur l'architecture E76 permet de gains de performance temporelle s'élevant jusqu'à 65% pour la différence d'image, sans perte de performance énergétique.

La Table 6.5 présente uniquement les résultats des versions (DD, naïve) et (DD, db) de la différence d'image sur l'architecture E76. Ces deux versions de la différence d'image respectivement l'utilisation de la C-SRAM sans et avec implémentation d'un SWP. Nous retenons la plage des images de 16 Kilo-octets à 16 Méga-octets. Nous pouvons conclure de ces résultats que l'utilisation du DMU pour implémenter un SWP offre des gains de performance temporelle pouvant s'élever à 65% lorsque le système mémoire de l'architecture est saturé. Ce gain de performance est obtenu, dans le cas de la différence d'image, sans baisse de performance énergétique.

6.3.2 Exploitation des ressources matérielles par les applications implémentées

L'objectif de cette évaluation est d'analyser la performance de nos applications d'intérêt selon le degré d'exploitation des ressources matérielles des différentes architectures modélisées. Ces analyses nous permettent de mettre au point des observations quand à la performance de nos implémentations logicielles, ainsi que les perspectives d'intégration architecturales de nos contributions. Dans le cadre de la présente sous-section, nous focaliserons les observations effectuées à la suite de ces analyses sur les perspectives et les aspects logiciels et architecturaux du point de vue de la mémoire C-SRAM et de son jeu global d'instructions.

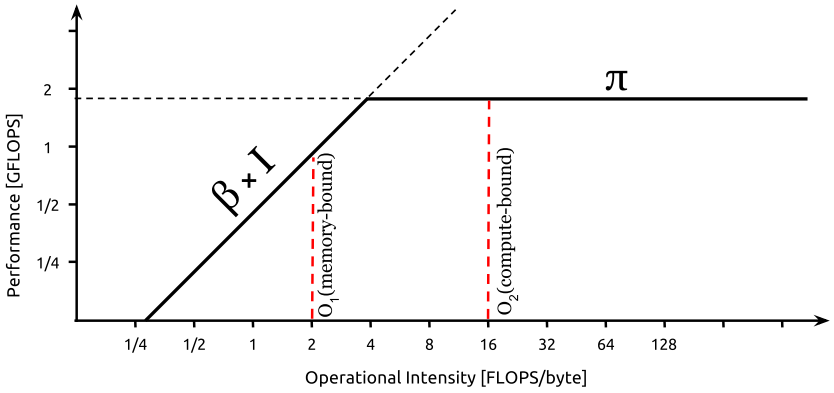


FIGURE 6.13 – Exemple graphique de roofline model d'une architecture non spécifiée, pour deux applications non spécifiées d'intensités arithmétique respectives O_1 et O_2. *Source : Wikimedia Commons.*

Notre méthodologie pour cette évaluation se base sur le ***Roofline Model*** [WWP09], un modèle numérique qui permet l'approximation des performances crêtes des architectures afin de positionner la performance des applications visées selon l'exploitation des ressources matérielles. La Figure 6.13 présente un exemple graphique de roofline pour une architecture de calcul non spécifiée. Sur cet exemple, le plafond de performance de l'architecture (la *roofline*) est représenté par la fonction suivante :

$$P = min(\pi, \beta \times I) \tag{6.5}$$

P est exprimé en nombre d'opérations par unité de temps – dans le présent exemple, en *Floating-Point Operations Per Second* (FLOPS). π correspond à la performance crête de calcul de l'architecture en FLOPS tandis que la fonction affine $\beta \times I$ est le produit de la bande passante maximale β par l'intensité arithmétique variable I. Ainsi, lors d'une représentation graphique, les unités des axes des abscisses et des ordonnées sont respectivement le *"nombre d'opérations par octet transféré"* et le *"nombre d'opérations par unité de temps"*. À partir de ce modèle, les applications exécutées sur l'architecture modélisée peuvent être catégorisées en deux classes :

— Les applications *memory-bound*, dont la performance est intrinsèquement limitée par la bande passante de l'architecture.

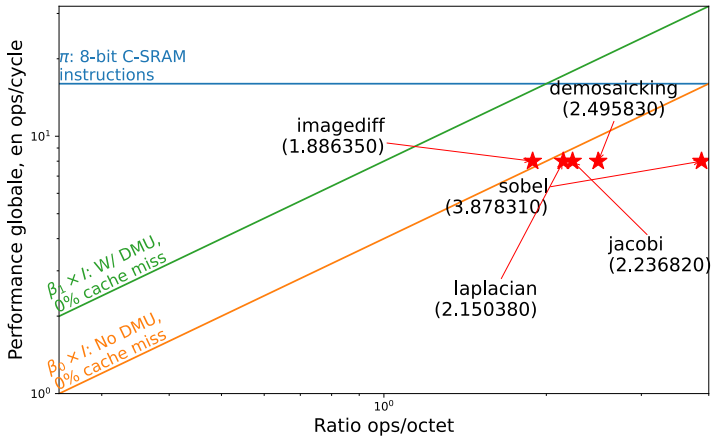
— Les applications *compute-bound*, dont la performance est intrinsèquement limitée par la performance de calcul de l'architecture.

Le Roofline Model permet de simplement établir la classification d'un jeu d'applications d'intérêt donné en utilisant une analyse graphique quant au positionnement de leurs intensités arithmétiques théoriques vis-à-vis d'une architecture donnée. *"Toute application positionnée en dessous du palier de bande passante crête est compute-bound, tandis que toute application positionnée en dessous du palier de performance de calcul crête est memory-bound"*. Sur le présent exemple, deux applications d'intensités arithmétique respectives O_1 et O_2 sont positionnées sur la représentation graphique du modèle. La première application est memory-bound, tandis que la seconde est compute-bound, par rapport à leurs exécutions respectives sur l'architecture modélisée en exemple.

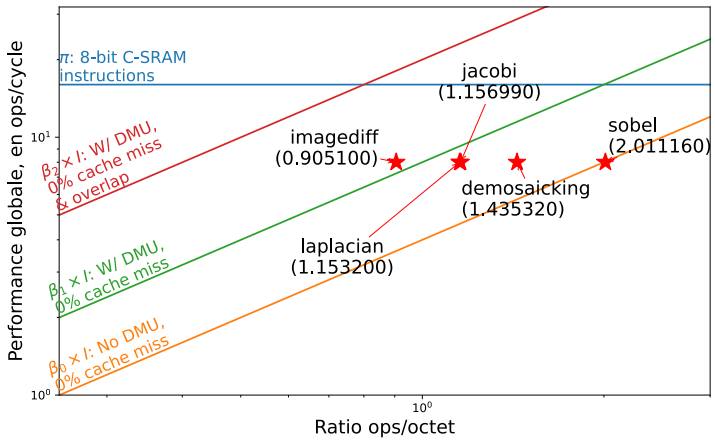
Le Roofline Model est ainsi un outil pratique pour évaluer rapidement le degré d'exploitation des ressources de calcul sur une architecture donnée par une implémentation logicielle donnée. Ce degré d'exploitation est déterminé par la qualité du code généré, dont les principaux responsables sont les programmeurs, mais également les interfaces logicielles tels que les compilateurs et générateurs de code. C'est ainsi que nous souhaitons employer le Roofline Model pour nos architectures modélisées, afin d'évaluer le degré d'exploitation de l'ensemble des composants.

Nous avons utilisé la méthodologie suivante pour établir les rooflines de nos trois architectures expérimentales :

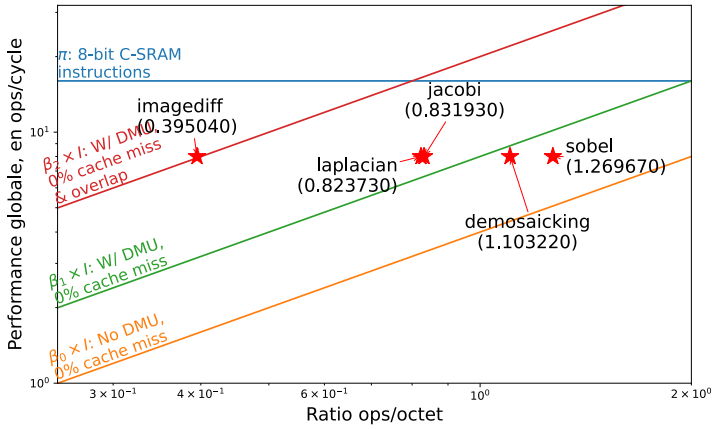
- Si le DMU n'est pas considéré comme utilisable par les implémentations logicielles, la bande passante à disposition est alors celle de la hiérarchie mémoire. Nous ne modélisons dans notre modèle aucune stratégie d'optimisation micro-architecturale tels que celles présentées lors du Chapitre 2 pour dissimuler la latence d'accès à la mémoire. En conséquence, la bande passante maximale dans ce cas d'usage est β_0 = 4 octets par cycle, partant de l'hypothèse que nous disposons d'un cache parfait (c'est-à-dire avec un taux de miss de 0%).
- Lorsque le contrôleur DMU est utilisé par les implémentations logicielles, ce dernier est interfacé par le biais d'un bus de 4 octets vers la mémoire principale, et un bus de 16 octets vers le stockage interne de la mémoire C-SRAM. Nous pouvons ainsi considérer deux seuils de bandes passantes :
  - β_1 = 8 octets par cycles, lors d'un accès de données par le contrôleur DMU sur la mémoire principale, et la lecture/écriture de données en par le HPU au niveau de sa hiérarchie mémoire.
  - β_2 = 20 octets par cycles, lors d'un accès de données par le contrôleur DMU sur la mémoire C-SRAM, et la lecture/écriture de données en par le HPU au niveau de sa hiérarchie mémoire.
- Le contrôleur DMU est modélisé, au sein de la simulation QEMU, pour être interfacé avec la C-SRAM par le biais d'un bus de 16 octets de largeur. Considérant que les bus servant d'interfaces entre chaque cache de la hiérarchie mémoire sont de 4 octets de largeur, la bande passante maximale absolue est β_2 = 4+16 = 20 octets par cycle HPU, pour l'ensemble des architectures expérimentales.



(a) Architecture E31.



(b) Architecture E76.



(c) Architecture U74.

FIGURE 6.14 – Ratio *ops/octet* des applications expérimentales, sur chacune des trois architectures de calcul expérimentales. Leurs rooflines sont graphiquement représentées pour rapidement estimer le degré d'exploitation des ressources de calcul de chaque application.

La Figure 6.14 présente nos résultats après l'application de notre méthodologie. Chaque sous-figure représente les ratios ops/octet des applications expérimentales – à l'exception de la multiplication matricielle - sur chaque architecture, pour des tailles de données maximales, afin de saturer l'ensemble du système mémoire des architectures. Les ratios ops/octet retenues pour chaque application sont celles parmi les meilleures versions d'implémentation générées à partir de notre précédente expérimentation (voir sous-section précédente).

Nous affichons également sur chaque sous-figure les seuils de performance précédemment définis des architectures expérimentales. Contrairement à l'exemple de roofline précédemment présenté lors de la Figure 6.13, nous utilisons un ratio ops/octet et une performance de calcul exprimée selon la cardinalité des calculs en C-SRAM, et le nombre de cycle d'horloges pour chaque architecture. La raison de ce choix est que la mémoire C-SRAM n'effectue pas de calcul à virgule flottante, il ne nous est donc pas possible d'utiliser le FLOPS comme métrique qualitative.

En observant la Figure 6.14a, nous pouvons observer que l'ensemble des implémentations logicielles sont exécutées avec un ratio ops/octet supérieure à 1 opération par cycle, ce qui signifie que les transferts de données sont effectivement bien amortis par les implémentations logicielles. Ces résultats se justifient par l'utilisation de techniques telles que le double-buffering, pour recouvrir temporellement les transferts de données et les calculs en C-SRAM, ainsi que le degré de parallélisme gagné en utilisant cette dernière. Une analyse visuelle de la figure nous montre également que l'ensemble des implémentations logicielles se comportent de façon compute-bound, car non-affectées par la limite de la bande passante. Nous pouvons ainsi conclure que l'augmentation du degré de parallélisme exploité par les applications, en utilisant de l'arithmétique 8-bit par exemple, bénéficierait à la performance de ces dernières.

La Figure 6.14b nous montre que les meilleures implémentations logicielles trouvées pour la majorité des applications présentent également un bon amorti du coût des transferts mémoires, à l'exception de la différence d'image. Nous expliquons ce phénomène par le fait que, dans le cas de la différence d'image, la charge arithmétique son implémentation est faible par rapport au coût du transfert des données (deux données transférées pour une opération de soustraction saturée) et la puissance de calcul à disposition pour ladite implémentation (opérations arithmétiques 16-bit).

Enfin, la Figure 6.14c présente une disparité dans les ratios ops/octet des implémentations logicielles évaluées. Une partie des applications disposent d'implémentations qui amortissent le coût des transferts de données – le dématriçage et le filtre de Sobel, tandis qu'une autre partie présente une charge arithmétique trop faible pour amortir ce coût – l'opérateur laplacien discret, l'itération de Jacobi et la différence d'image. La différence d'image est l'application dont les résultats de l'implémentation logicielle sur l'architecture U74 nous sont les plus remarquables, car elle met en lumière sa nature memory-bound. Elle est donc, pour cette architecture expérimentale, l'application qui ne bénéficierait pas d'amélioration de performance par le seul changement dans la manière dont elle exploite les performances de calcul de la C-SRAM. Les autres applications pourraient bénéficier de meilleures performances par ce biais.

Lorsque nous analysons la plage des ratios ops/octet mesurées pour l'ensemble des implémentations logicielles, pour chaque architecture, nous pouvons ainsi remarquer que l'intégration de la C-SRAM et du contrôleur DMU utilisé pour nos expérimentations présente de meilleurs résultats, en termes de possibilité d'exploitation des ressources, sur des architectures frugales telles que l'architecture E31, que sur des architectures plus performantes comme la E76 et la

U74.

6.3.3 Impact de la génération dynamique de code sur la performance applicative

L'objectif de cette évaluation est d'évaluer la performance des solutions de compilation dynamique fournies par l'environnement Hybrogen, comparé à un flot de compilation classique, et donc statique. Pour ce faire, nous restreignons l'espace d'exploration logiciel à une seule application, ici la multiplication matricielle. Nous choisissons cette application car elle présente, parmi les cinq retenues, la plus grande complexité algorithmique et, une fois optimisée avec du partitionnement de boucles, la plus forte intensité arithmétique théorique. Nous utilisons parmi les trois architectures modélisées, celle avec le moins de complexité matérielle disponible, c'est-à-dire celle basée sur le cœur de calcul SiFive E31.

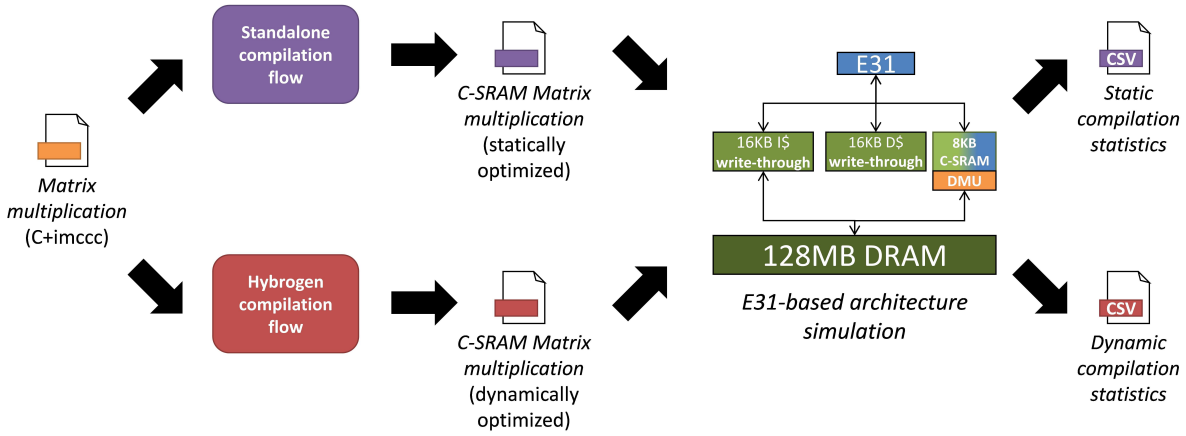


FIGURE 6.15 – Présentation de la méthodologie expérimentale pour évaluer l'impact de la compilation dynamique sur la performance.

La Figure 6.15 présente notre méthodologie générale. L'outil représentant la compilation dynamique est Hybrogen tandis que la compilation statique est effectuée par le compilateur `GCC` en version 10.2.0. Les deux flots de compilation utilisent cette version de `GCC`, avec les mêmes niveaux d'optimisation, c'est-à-dire `-O3`. Le code compilé de façon uniquement statique nous servira de référence pour le reste de cette sous-section.

	Compilation statique	Compilation dynamique
Taille totale du code	1,496 KB	13,5 KB (statique) +1,0 KB (dynamique)

TABLE 6.6 – Comparaison des tailles de code sur la version statique et a version spécialisée pour la compilation dynamique de code.

Nous souhaitons tout d'abord analyser comment évolue l'impact de la compilation de la multiplication matricielle sur la taille du code, une fois intégralement compilé. Nous utilisons

l'outil `objdump`, qui permet d'obtenir de nombreuses informations, y compris la taille de sections de code particulières. Nous obtenons par cette méthode les résultats présentés à la Table 6.6.

La version compilée exclusivement de façon statique possède une fonction de multiplication matricielle de 1496 octets. La version compilée pour intégrer la compilation dynamique d'une partie du code, quant à elle, possède une empreinte mémoire de 13,5 Kilo-octets. Cette empreinte est due au surcoût des générateurs d'instructions pour la génération dynamique du code. Il faut également considérer la section de code qui est allouée pour contenir le code généré lors de l'exécution par les générateurs d'instructions. L'empreinte totale du code est alors de 14,5 Kilo-octets. Bien que cette expansion de code semble conséquente lorsqu'elle est comparée à la version statique, il est néanmoins important de poursuivre notre étude comparative pour évaluer les gains apportés par la compilation dynamique sur la base d'autres métriques.

Ces gains doivent être évalués lors de l'exécution des versions statique et dynamique de l'application pour évaluer leur comportement sur l'architecture expérimentale ciblée. Nous exécutons le code pour différentes instances de tailles de matrices carrées. Les dimensions de ces matrices varient de 4×4 (soit 64 octets) à 512×512 (soit 1 Mega octet).

	Static code-gen	Dynamic codegen
64 B	0.01449	0.02381
1 KB	0.00086	0.00424
4 KB	0.00014	0.00078
16 KB	0.00002	0.00011
64 KB	0.00000	0.00001
256 KB	0.00000	0.00000
1 MB	0.00000	0.00000

TABLE 6.7 – Taux de miss de lecture vers le cache L1 instruction.

L'étude du taux de miss de lecture sur le cache d'instructions lors de l'exécution respective des deux versions nous montrent que pour des matrices 4×4, le taux de miss de lecture est de $1,45\%$ pour la version statique et de $2,38\%$ pour la version dynamiquement compilée (Table 6.7. Nous jugeons que chacun de ces taux, évalués à part, sont acceptables compte tenu de la taille du code, des dimensions du cache d'instructions ciblé ainsi que de la quantité d'instructions de branchement présent dans le code. Il est cependant intéressant de noter que la version dynamiquement compilée induit un surcoût de chargement d'instructions négligeable, qui est amorti dès le calcul de matrices de taille 1 Kilo-octet (taux de miss $< 0.5\%$). Cela est dû au fait que la séquence de génération dynamique de code n'est appelée qu'une seule fois, et laisse ensuite la place au code de multiplication matricielle. Par le fonctionnement du cache d'instruction, l'espace de stockage de ce dernier n'est donc pas occupé de façon concurrente par le générateur de code et le code généré.

Nous analysons la qualité d'exécution de la multiplication matricielle dynamiquement compilée comparée à la multiplication matricielle statique. Pour définir la qualité d'exécution, nous utilisons les latences d'exécution totale (en cycles d'horloge) et la consommation énergétique totale (en pico-Joules) de chaque version de multiplication matricielle.

	Latency (clk.)		Energy cost (pJ)	
	Static code-gen	Dynamic codegen	Static code-gen	Dynamic code-gen
64 B	1.163400e+04	3.794830e+05	1.141988e+07	3.779088e+08
1 KB	2.502200e+04	4.039420e+05	2.411702e+07	3.911724e+08
4 KB	6.000900e+04	4.697990e+05	5.515842e+07	4.288651e+08
16 KB	3.334530e+05	8.595160e+05	3.214229e+08	6.554212e+08
64 KB	2.107836e+06	3.261989e+06	2.021142e+09	2.029851e+09
256 KB	1.558715e+07	2.229679e+07	1.428558e+10	1.245721e+10
1 MB	1.144774e+08	1.712741e+08	1.039473e+11	9.262726e+10

TABLE 6.8 – Statistiques globales d'exécutions des versions de multiplication matricielle sur C-SRAM.

La Table 6.8 présente les résultats de cette analyse. Nous discuterons tout d'abord des latences d'exécutions. Nous pouvons observer que pour des matrices de tailles minimales, l'écart de performance entre la multiplication matricielle statique et dynamique est remarquablement élevé. La version dynamique est plus longue en nombres de cycles d'horloge que la version statique par un facteur d'environ ×32. Néanmoins, au fur et à mesure que les tailles de matrices augmentent, l'écart de performance entre la multiplication matricielle statique et dynamique se réduit considérablement. Pour des matrices d'un Méga-octet, le surcoût de performance de la génération de code dynamique de la multiplication matricielle n'est que de 10%, par rapport à l'implémentation statiquement compilée. Ce résultat est d'autant plus remarquable qu'il faut rappeler que la version de Hybrogen utilisée lors des expérimentations possède un jeu d'optimisations bien plus restreint qu'un compilateur gradé industriel tel que `GCC`, et les générateurs dynamiques d'instructions implémentés ne font pas d'optimisations telles que celles présentées dans le chapitre 4. Ainsi, nous pouvons d'ores et déjà affirmer que de nombreuses opportunités d'amélioration de ces résultats sont envisageables. Nous discuterons dans une section ultérieure de ces perspectives en plus grand détail.

	Compilation statique	Compilation dynamique
Ratio instr. HPU / instr. C-SRAM	1 pour 11	1 pour 16

TABLE 6.9 – Comparaison du débit d'instructions C-SRAM par rapport aux instructions HPU.

Cette non-optimalité du code généré par Hybrogen a également un impact direct sur l'efficacité de la programmation de la C-SRAM. Comme nous pouvons le voir à la Table 6.9, la multiplication matricielle statique présente un débit de génération d'instructions C-SRAM plus important que la version dynamique. La raison est que les instructions C-SRAM sont générées sous la forme de requêtes d'écritures en mémoires de mots de 32 bits. `GCC` ne requiert donc aucune modification pour utiliser le plein potentiel de ses optimisations pour améliorer ces requêtes d'écritures, et ainsi la programmation de la C-SRAM. Cependant, l'utilisation de Hybrogen ouvre la porte vers des perspectives d'optimisations dynamiques de la programma-

tion de la C-SRAM, via l'application d'optimisations de code au niveau de la génération des instructions mais également des instructions elle-même.

6.3.4 Observations

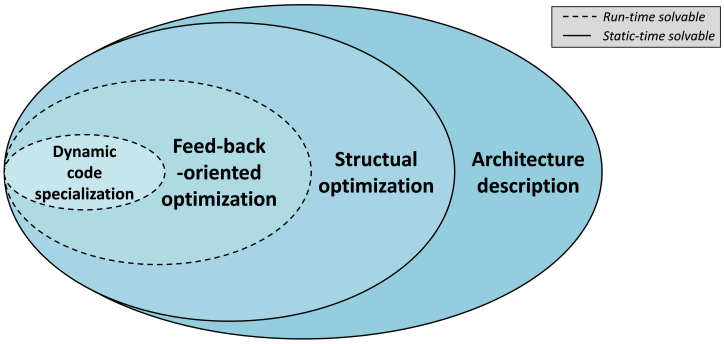


FIGURE 6.16 – Le spectre des optimisations de code réalisables sur le cycle de vie d'une application, de la conception au moment d'exécution.

Nous avons présenté au cours de ce chapitre une multitude d'exérimentations pour évaluer de façon qualitative l'impact de nos contributions.

Nous avons évalué la ***compilation dynamique de code***, qui vise à générer des instructions lors de l'exécution tout en appliquant des optimisations de code autrement insolvables lors de la compilation statique des applications.

Une seconde approche intéressante à étudier serait le ***feedback-oriented optimization***, qui permet par le biais d'analyses de trace d'exécution, de déterminer des paramètres de programmation optimaux selon un jeu de métriques donné. L'utilisation de ces deux paradigmes d'optimisation permet d'étendre la portée de l'optimisation de code au-delà de la partie inerte du cycle de vie des applications, pour l'étendre au temps d'exécution et au-delà (Figure 6.16).

De précédents travaux ont établi des pistes de recherche d'intérêt pour poursuivre cette direction. Les travaux présentés dans [Lom14] étudiait la spécialisation dynamique de code à l'aide de `deGoal`, prédecesseur de Hybrogen, et présentait `Kahuna`, un outil qui permet la spécialisation dynamique d'une application donnée par la sélection d'*instances*, des implémentations variées de ladite application. Dans le cadre de [LY20], le modèle de programmation HARDSI est présenté pour permettre aux programmeurs la mise en place d'un flot d'optimisation *feedback-oriented*. Ces travaux pourraient être employés pour mettre en place un flot d'automatisation pour saisir les opportunités présentées par l'optimisation feedback-oriented de l'utilisation du contrôleur DMU.

À notre connaissance, aucune autre contribution de la littérature n'a étudié la synergie entre ces différents paradigmes pour les évaluer de façon concrète. Nous concluons que l'optimisation de code multi-paradigme présente des perspectives encourageantes vers de nouvelles pistes de recherche et de développement.

Conclusion

Au cours de ce chapitre, nous avons présenté notre méthodologie d'évaluation du contrôleur DMU et du modèle de programmation de la C-SRAM. Nous avons effectué l'évaluation de six applications dans le domaine du traitement d'image et de la Computer Vision sur différentes architectures matérielles. La modélisation et la simulation des architectures ont été effectuées sur la base de QEMU, pour évaluer l'exécution des applications.

Nous concluons que l'utilisation de la compilation dynamique et du Software Pipelining présente des perspectives prometteuses, qui pourraient bénéficier de l'optimisation dynamique de code.

Conclusion générale

Contributions

Les objectifs initiaux de la présente thèse étaient de concevoir un modèle de programmation pour une architecture de calcul proche-mémoire, plus particulièrement la SRAM Computationelle (C-SRAM), ainsi qu'une méthodologie d'évaluation de systèmes intégrant de la mémoire C-SRAM au sein de leurs hiérarchies mémoires. Ces prémisses ont finalement donné lieu à une démarche de co-conception, qui a maintenu tout le long du temps de développement de nos contributions la cohérence entre les architectures et les applications visées.

Les contributions de la présente thèse furent :

- La spécification du *Data-locality Management Unit*, un contrôleur mémoire dédié au transfert et la réorganisation de données de stencil. Cette spécification nous a demandé de traiter des problématiques quand à au dimensionnement de l'interface de programmation d'un tel composant.
- L'implémentation d'IMCCC, un modèle de programmation et compilateur dédié à la mémoire C-SRAM, ainsi qu'au contrôleur DMU. Ce modèle de programmation a posé des réflexions quant à l'expression à haut niveau d'algorithmes élémentaires tels que les codes de stencil, et leur support auprès de nœuds de calcul hétérogènes.
- Le support de la compilation dynamique de code pour la mémoire C-SRAM, par le biais de l'environnement de compilation Hybrogen. Nous avons montré que ce support pouvait être porté jusqu'à un modèle de programmation haut-niveau tel que celui proposé par IMCCC grâce à une méthodologie d'inter-opération entre IMCCC et HybroLang, le langage spécialisé de Hybrogen.
- L'implémentation d'un flot de modélisation et de simulation de systèmes de calcul. Ce flot de simulation nous a permis de valider et d'évaluer des propositions logicielles et

matérielles impliquant la mémoire C-SRAM et le contrôleur DMU, en avance de phase d'implémentation physique.

Il existe, à notre connaissance, peu de travaux de thèses qui ont et présentent l'opportunité de travailler sur la jonction du développement logiciel et de la conception matérielle. Les projets de recherche sur lesquels ces contributions ont été basées sont tout aussi intéressants par la voie inédite qu'ils prennent. Il existe en effet très peu de travaux récents sur l'étude de la compilation dynamique, bien que les résultats obtenus soient prometteurs. Nous avons également eu l'occasion d'étudier l'impact de ce paradigme de génération de code sur une architecture émergente, à savoir la C-SRAM. Il existe, à notre connaissance, quasiment aucune contribution de l'art antérieur qui a émis de résultats quant à l'utilisation de la compilation dynamique pour les architectures de calcul hétérogènes.

Cette opportunité a été offerte par la nature pluridisciplinaire du LFIM – le laboratoire d'accueil de cette thèse, mais aussi de l'équipe en son sein, qui est composé de chercheurs en informatique et en conception électronique. Ainsi, la thèse a pu être réalisée en parallèle de la conception d'un circuit-test, sur lequel les contributions pourront être vérifiées et validées.

Nous pensons que les questions scientifiques abordées au cours de la thèse peuvent donner lieu à une méthodologie systématique pour la co-conception de systèmes logiciels et matériels efficaces, à l'ère des architectures de calcul fortement hétérogènes et émergentes.

Perspectives introduites à la suite des contributions

Auto-tuning des applications pour les architectures de calcul proche-mémoire

Nos expérimentations autour de la mémoire C-SRAM nous ont permis de déterminer que l'utilisation d'un contrôleur mémoire spécialisé tel que le DMU apporte un gain de performance important en termes de consommation énergétique et de temps d'exécution. Nous avons déterminé un intérêt quand à l'auto-tuning des applications C-SRAM pour maximiser la performance des applications selon un certain jeu de critères paramétrables. Cet intérêt peut être étendu au paradigme architectural du calcul proche mémoire, et mis en place par le biais d'un flot de compilation et/ou de programmation par feedback, qui automatiserait les phases d'analyse de performance d'exécution pour paramétrer les applications IMC de façon optimale.

Exploration logicielle des applications

Nous avons présenté, lors de la présente thèse, l'implémentation de six applications des domaines du traitement d'image, de la Computer Vision, ainsi que de l'algèbre linéaire. De nombreuses autres routines, standardisées selon des spécifications telles qu'OpenCV ou OpenVX, restent encore à implémenter. L'art antérieur propose, par exemple, de nombreuses solutions de traitement d'image basées sur le filtrage morphologique [LMDM09] qui bénéficieraient des contributions apportées dans la présente thèse. Nous envisageons également de poursuivre notre exploration logicielle vers le domaine de l'IA et plus spécifiquement du Deep Learning. Les travaux de recherche poussent les réseaux de neurones artificiels vers des modèles de plus en plus compacts et requérant de moins en moins de précision de calcul, tels que les modèles

SqueezeNet [IHM+16] et MobileNet [HSC+19], mais également la classe des réseaux de neurones ternaires [ALPBP17].

Les travaux entrepris sur Hybrogen seront davantage développés pour étudier l'intégration de passes d'optimisation dynamique. L'utilisation de la compilation dynamique par le biais de Hybrogen reste une approche intéressante pour le développement d'applications standardisées pouvant être spécialisées lors de l'exécution.

Exploration de l'intégration système de la mémoire C-SRAM

Notre flot de modélisation et de simulation de systèmes C-SRAM nous a permis d'effectuer de l'exploration architecturale focalisée sur l'intégration de la C-SRAM au sein d'une hiérarchie mémoire hôte. Nous avons pu observer que, par le modèle d'interface actuellement spécifié pour la C-SRAM et le DMU, les gains de performance apportés par la C-SRAM sur certaines applications peuvent varier. Nous observons ainsi un intérêt à utiliser le modèle de simulation en avance de phase d'implémentation pour effectuer de l'exploration architecturale selon un jeu d'applications visées. Nous prévoyons, parmi d'autres, les situations suivantes :

— Modélisation d'architectures monocœur multi- C-SRAM. Dans ces scénarios, le modèle de programmation et d'interface présenté au cours de la présente thèse est préservé tout en offrant un degré de vectorisation de code supérieur, et potentiellement paramétrable. Des questions se posent, cependant quant au transfert et la réorganisation de données au sein des différentes unités de C-SRAM. Cette exploration architecturale sera réalisée dans un futur proche.

— Modélisation d'architectures multicœurs multi- C-SRAM. De telles architectures incluent des architectures SMT, mais également des architectures DSM composés de plusieurs CPUs, eux-mêmes en possession d'une unité de mémoire C-SRAM. Le modèle de programmation proposé dans la présente thèse pourrait être réutilisé grâce à son inter-opération avec d'autres modèles de programmation spécialisés pour la programmation multicœurs et multisystèmes, tels qu'OpenMP ou encore MPI. Une telle exploration de co-conception logicielle/matérielle est envisageable à moyen terme.

— Interface des mémoires C-SRAM avec la hiérarchie mémoire au niveau des mémoires cache. Une telle intégration de la C-SRAM au sein d'une architecture hôte pourrait permettre d'amortir la latence d'accès de la mémoire C-SRAM, mais nécessite la mise en place des mécanismes de cohérence de la mémoire dont le coût est à estimer au niveau architectural. Une telle exploration est envisageable au long terme.

Bibliographie

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research : A view from berkeley. 2006.

[ALPBP17] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient ai applications. In *2017 international joint conference on neural networks (IJCNN)*, pages 2547–2554. IEEE, 2017.

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. *ACM SIGPLAN Notices*, 31(5) :149–159, 1996.

[Arm] Arm. Making Temporal Prefetchers Practical : The MISB Prefetcher - Research Articles - Arm Research - Arm Community.

[ASL+19] Hameeza Ahmed, Paulo C Santos, João PC Lima, Rafael F Moura, Marco AZ Alves, Antônio CS Beck, and Luigi Carro. A compiler for automatic selection of suitable processing-in-memory instructions. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 564–569. IEEE, 2019.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2) :1–7, 2011.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[BGOS12] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*, pages 1–7. IEEE, 2012.

[Bha02] Jayaram Bhasker. *A SystemC primer*, volume 2. Star Galaxy Publishing Allentown, 2002.

[BM06] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1) :1–es, 2006.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[Bro82] Allan G Bromley. Charles babbage's analytical engine, 1838. *Annals of the History of Computing*, 4(3) :196–217, 1982.

[BRR+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu : A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[CCL+14] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A Endo, and Rémy Gauguey. degoal a tool to embed dynamic code generators into applications. In *International Conference on Compiler Construction*, pages 107–112. Springer, 2014.

[CCWZ18] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda : Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[CDM+05] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. An empirical study of programming language trends. *IEEE software*, 22(3) :72–79, 2005.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM} : An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[Cop04] B Jack Copeland. Colossus : its origins and originators. *IEEE Annals of the History of Computing*, 26(4) :38–45, 2004.

[DBH+21a] Alain Denzler, Rahul Bera, Nastaran Hajinazar, Gagandeep Singh, Geraldo F. Oliveira, Juan Gómez-Luna, and Onur Mutlu. Casper : Accelerating stencil computation using near-cache processing. *CoRR*, abs/2112.14216, 2021.

[DBH+21b] Alain Denzler, Rahul Bera, Nastaran Hajinazar, Gagandeep Singh, Geraldo F Oliveira, Juan Gómez-Luna, and Onur Mutlu. Casper : Accelerating stencil computation using near-cache processing. *arXiv preprint arXiv :2112.14216*, 2021.

[DCMK21] Julie Dumas, Henri-Pierre Charles, Kévin Mambu, and Maha Kooli. Dynamic compilation for transprecision applications on heterogeneous platform. *Journal of Low Power Electronics and Applications*, 11(3) :28, 2021.

[EBG+20] Mona Ezzadeen, D Bosch, Bastien Giraud, S Barraud, J-P Noel, Didier Lattard, Joris Lacord, Jean-Michel Portal, and François Andrieu. Ultrahigh-density 3-d vertical rram with stacked junctionless nanowires for in-memory-computing applications. *IEEE Transactions on Electron Devices*, 67(11) :4626–4630, 2020.

[ENK+21] Valentin Egloff, Jean-Philippe Noel, Maha Kooli, Bastien Giraud, Lorenzo Ciampolini, Roman Gauchi, César Fuguet, Éric Guthmuller, Mathieu Moreau, and Jean-Michel Portal. Storage class memory with computing row buffer : A design space exploration. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2021.

[FDB+10] R Alves Fonseca, Luigi Dilillo, Alberto Bosio, Patrick Girard, Serge Pravossoudovitch, Arnaud Virazel, and Nabil Badereddine. Analysis of resistive-bridging defects in sram core-cells : A comparative study from 90nm down to 40nm technology nodes. In *2010 15th IEEE European Test Symposium*, pages 132–137. IEEE, 2010.

[FJT12] Ylva Fernaeus, Martin Jonsson, and Jakob Tholander. Revisiting the jacquard loom : threads of history and current patterns in hci. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1593–1602, 2012.

[Fly66] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.

[Fou] GNU Foundation. Using the GNU Compiler Collection (GCC) : Common Function Attributes.

[FP16] Davide Ferraretto and Graziano Pravadelli. Simulation-based fault injection with qemu for speeding-up dependability analysis of embedded software. *Journal of Electronic Testing*, 32(1) :43–57, 2016.

[FR96] Michael J Flynn and Kevin W Rudd. Parallel architectures. *ACM computing surveys (CSUR)*, 28(1) :67–70, 1996.

[FW87] José AB Fortes and Benjamin W Wah. Systolic arrays : A survey of seven projects. *Computer*, 20(07) :91–91, 1987.

[GEK+20] Roman Gauchi, Valentin Egloff, Maha Kooli, J-P Noel, Bastien Giraud, Pascal Vivet, Subhasish Mitra, and H-P Charles. Reconfigurable tiles of computing-in-memory sram architecture for scalable vectorization. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 121–126, 2020.

[Get11] Pascal Getreuer. Malvar-he-cutler linear image demosaicking. *Image Processing on Line*, 1 :83–89, 2011.

[GFJ+20] S Guo, Yongliang Feng, Jérôme Jacob, Florian Renard, and Pierre Sagaut. An efficient lattice boltzmann method for compressible aerodynamics on d3q19 lattice. *Journal of Computational Physics*, 418 :109570, 2020.

[GGL12] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04) :1250010, 2012.

[GK17] Fatih Gurcan and Cemal Kose. Analysis of software engineering industry needs and trends : Implications for education. *International Journal of Engineering Education*, 33(4) :1361–1368, 2017.

[GLD10] Daniel Große, Hoang M Le, and Rolf Drechsler. Proving transaction and system-level properties of untimed systemc tlm designs. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 113–122. IEEE, 2010.

[GLW98] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pages 106–111. IEEE, 1998.

[GM] Alain Greiner and E Martin. The soclib project.

[GMP+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. Dyc : an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1-2) :147–199, 2000.

[God21] Akira Goda. Recent progress on 3d nand flash technologies. *Electronics*, 10(24) :3156, 2021.

[GRC+20] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. Pulp-nn : accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A*, 378(2164) :20190155, 2020.

[Gro17] Khronos Vision Working Group. The OpenVX Specification version 1.2, October 2017.

[GRS07] Christian Grossmann, Hans-Görg Roos, and Martin Stynes. *Numerical treatment of partial differential equations*, volume 154. Springer, 2007.

[HAMR22] Ala Hijazi, Ahmad Al-Masri, and Nathir Rawashdeh. On the use of bayer sensor color cameras in digital image correlation. In *2022 11th International Symposium on Signal, Image, Video and Communications (ISIVC)*, pages 1–7. IEEE, 2022.

[HBD+14] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom :

compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4) :144–1, 2014.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. Ddm-a cache-only memory architecture. *Computer*, 25(9) :44–54, 1992.

[HNKK17] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. Cairo : A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4) :1–25, 2017.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[HP11] John L Hennessy and David A Patterson. *Computer architecture : a quantitative approach*. Elsevier, 2011.

[HSC+19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[IHM+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet : Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv :1602.07360*, 2016.

[Int] Intel. oneAPI : A New Era of Heterogeneous Computing.

[IRC11] Mark Ilg, Jonathan Rogers, and Mark Costello. Projectile monte-carlo trajectory analysis using a graphics processing unit. In *AIAA atmospheric flight mechanics conference*, page 6266, 2011.

[Jou90] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI) :364–373, 1990.

[KAH07] Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic compilation : the benefits of early investing. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 94–104, 2007.

[KCT+18] Maha Kooli, Henri-Pierre Charles, Clement Touzet, Bastien Giraud, and Jean-Philippe Noel. Smart instruction codes for in-memory computing architectures compatible with standard sram interfaces. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1634–1639. IEEE, 2018.

[KD17] K Ashok Kumar and P Dananjayan. A survey for silicon on chip communication. *Indian Journal of Science and Technology*, 10(1) :1–10, 2017.

[KGSC01] Chandra J Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software : Practice and Experience*, 31(8) :717–738, 2001.

[KP05] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming : A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2) :83–137, 2005.

[KR07] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2) :203–215, 2007.

[KWL16] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573. IEEE, 2016.

[LAB+20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir : A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv :2002.11054*, 2020.

[LEHZ+14] Lionel Lacassagne, Daniel Etiemble, Ali Hassan Zahraee, Alain Dominguez, and Pascal Vezolle. High level transforms for simd and low-level computer vision algorithms. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/-Vector processing*, pages 49–56, 2014.

[LLO+19] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. Delta : Gpu performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 293–303. IEEE, 2019.

[LMDM09] Lionel Lacassagne, Antoine Manzanera, Julien Denoulet, and Alain Mérigot. High performance motion detection : some trends toward new embedded architectures for vision systems. *Journal of Real-Time Image Processing*, 4(2) :127–146, 2009.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla : A unified graphics and computing architecture. *IEEE micro*, 28(2) :39–55, 2008.

[Loh08] Gabriel H Loh. 3d-stacked memory architectures for multi-core processors. *ACM SIGARCH computer architecture news*, 36(3) :453–464, 2008.

[Lom14] Victor Lomüller. *Générateur de code multi-temps et optimisation de code multi-objectifs*. PhD thesis, Grenoble, 2014.

[LPAA+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator : Version 20.0+. *arXiv preprint arXiv :2007.03152*, 2020.

[LWT+19] Jie Li, Xi Wang, Antonino Tumeo, Brody Williams, John D Leidel, and Yong Chen. Pims : a lightweight processing-in-memory accelerator for stencil computations. In *Proceedings of the International Symposium on Memory Systems*, pages 41–52, 2019.

[LWZC17] Zhen Li, Yuqing Wang, Tian Zhi, and Tianshi Chen. A survey of neural network accelerators. *Frontiers of Computer Science*, 11(5) :746–761, 2017.

[LY20] Sid Lakhdar and Riyane Yacine. *Méthodologie pour l'optimisation logicielle de structures de données pour les architectures hautes performances à mémoires complexes*. PhD thesis, Université Grenoble Alpes, 2020.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0 : A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, 147, 2009.

[MCDK21] Kévin Mambu, Henri-Pierre Charles, Julie Dumas, and Maha Kooli. Instruction set design methodology for in-memory computing through qemu-based system emulator. In *32rd International Workshop on Rapid System Prototyping*, 2021.

[MCE00] Markus Mock, Craig Chambers, and Susan J Eggers. Calpa : A tool for automating selective dynamic compilation. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 291–302. IEEE, 2000.

[MCKD22] Kévin Mambu, Henri-Pierre Charles, Maha Kooli, and Julie Dumas. Towards integration of a dedicated memory controller and its instruction set to improve performance of systems containing computational sram. *Journal of Low Power Electronics and Applications*, 12(1) :18, 2022.

[MHC04] Henrique Malvar, Li-wei He, and Ross Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. volume 3, pages iii – 485, 06 2004.

[Mit16] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2) :1–35, 2016.

[MPTC+20] Ivan Miro-Panades, Benoit Tain, Jean-Frédéric Christmann, David Coriat, Romain Lemaire, Clement Jany, Baudouin Martineau, Fabrice Chaix, Anthony Quelen, Emmanuel Pluchart, et al. Samurai : A 1.7 mops-36gops adaptive versatile iot node with 15,000× peak-to-idle power reduction, 207ns wake-up time and 1.3 tops/w ml efficiency. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2. IEEE, 2020.

[MR13] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[MRH+01] Wolfgang Mueller, Juergen Ruf, Dirk Hoffmann, Joachim Gerlach, Thomas Kropf, and Wolfgang Rosenstiehl. The simulation semantics of systemc. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 64–70. IEEE, 2001.

[Mun09] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage : Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News*, 43(1) :429–443, 2015.

[NK12] Bill Neifert and Rob Kaye. High Performance or Cycle Accuracy ? Technical report, ARM, 2012.

[NPG+20] J-P Noel, Manuel Pezzin, Roman Gauchi, J-F Christmann, Maha Kooli, H-P Charles, Lorenzo Ciampolini, Mariam Diallo, Florent Lepin, Benjamin Blampey, et al. A 35.6 tops/w/mm^2 3-stage pipelined computational sram with adjustable form factor for highly data-centric applications. *IEEE Solid-State Circuits Letters*, 3 :286–289, 2020.

[NSAR20] Salwa Yasmeen Neyaz, Itisha Saxena, Naushad Alam, and Syed Atiqur Rahman. Fpga and asic implementation and comparison of multipliers. In *2020 International Symposium on Devices, Circuits and Systems (ISDCS)*, pages 1–4. IEEE, 2020.

[NSS+16] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks : Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[NVI] NVIDIA. CUDA C++ Programming Guide.

[OLG+07] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[Opea] OpenCV. OpenCV : Laplace Operator.

[Opeb] OpenVX. The OpenVX Specification : Sobel 3x3.

[Ore00] Nir Oren. A survey of prefetching techniques. *Technical report, July 2000*, 2000.

[Pan01] Preeti Ranjan Panda. Systemc : a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, 2001.

[Par13] Lucas S Parobek. Research, development and testing of a fault-tolerant fpga-based sequencer for cubesat launching applications. Technical report, NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2013.

[PBB+09] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, R Ramanujam, and P Sadayappan. *Hybrid iterative and model-driven optimization in the polyhedral model*. PhD thesis, INRIA, 2009.

[PHG11] Marcel Pockrandt, Paula Herber, and Sabine Glesner. Model checking a systemc/tlm design of the amba ahb protocol. In *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 66–75. IEEE, 2011.

[PHO+14] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu : A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1) :34–36, 2014.

[RKA+11] Nursalasawati Rusli, Erwan Hafizi Kasiman, Kueh Beng Hong Ahmad, Airil Yasreen Mohd Yassin, and Norsarahaida Amin. Numerical computation of a two-dimensional navier-stokes equation using an improved finite difference method. *MATEMATIKA : Malaysian Journal of Industrial and Applied Mathematics*, pages 1–9, 2011.

[RKAS+17] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide : Decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1) :106–115, 2017.

[Sch10] Andres Schäfer. Geometrische zerlegung. In Dietmar Fey, editor, *Grid-Computing : Eine Basistechnologie für Computational Science*, pages 425–465. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[SCR+17] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race ? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[SDH+19] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Sander Stuijk, and Henk Corporaal. Narmada : Near-memory horizontal diffusion accelerator for scalable stencil computations. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 263–269. IEEE, 2019.

[SDH+20] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Nero : A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 9–17. IEEE, 2020.

[SiFa] Inc SiFIve. HF105 Datasheet.

[SiFb] Inc SiFive. SiFive FE310-G000 Manual v2p3.

[Sin14] Nishu Singla. Motion detection based on frame difference method. *International Journal of Information & Computation Technology*, 4(15) :1559–1565, 2014.

[SKH08] Erno Salminen, Ari Kulmala, and Timo D Hamalainen. Survey of network-on-chip proposals. *white paper, OCP-IP*, 1 :13, 2008.

[SKKP22] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. Rise & shine : Language-oriented compiler design. *arXiv preprint arXiv :2201.03611*, 2022.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3) :473–530, 1982.

[Sta01] Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. Technical report, August 2001.

[SW83] JM Speiser and HJ Whitehouse. A review of signal processing with systolic arrays. *Real-Time Signal Processing VI*, 431 :2–6, 1983.

[TLNA21] Anton V Trusov, Elena E Limonova, Dmitry P Nikolaev, and Vladimir V Arlazarov. p-im2col : Simple yet efficient convolution algorithm with flexibly controlled memory overhead. *IEEE Access*, 9 :168162–168184, 2021.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular automata machines : a new environment for modeling*. MIT press, 1987.

[Uch13] Seiichi Uchida. Image processing and recognition for biological images. *Development, growth & differentiation*, 55(4) :523–549, 2013.

[UPD+20] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. Hardware-oblivious simd parallelism for in-memory column-stores. In *CIDR*, 2020.

[VL96] Steve VanderWiel and David J Lilja. A survey of data prefetching techniques. In *Procs. of the 23rd International Symposium on Computer Architecture*. Citeseer, 1996.

[VL00] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2) :174–199, 2000.

[VN93a] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4) :27–75, 1993.

[VN93b] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4) :27–75, 1993.

[VR+09] Peter Van Roy et al. Programming paradigms for dummies : What every programmer should know. *New computational paradigms for computer music*, 104 :616–621, 2009.

[WDL20] Waseem Waheed, Guang Deng, and Bo Liu. Discrete laplacian operator and its applications in signal processing. *IEEE Access*, 8 :89692–89707, 2020.

[Whe19] Bob Wheeler. WD Rolls Its Own RISC-V Core. *The Linley Group Microprocessor Report*, February 2019 :3, 2019.

[WST+12] Sandra Wienke, Paul Springer, Christian Terboven, et al. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline : an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4) :65–76, 2009.

[ZKCS02] Cesar A Zeferino, Márcio E Kreutz, Luigi Carro, and Altamiro A Susin. A study on communication issues for systems-on-chip. In *Proceedings. 15th Symposium on Integrated Circuits and Systems Design*, pages 121–126. IEEE, 2002.

Table des figures

Liste des tableaux

Acronymes

ALU Arithmetic-Logic Unit.
API Application-Programming Interface.
ASCC Automated Sequence-Controlled Calculator.
ASIC Application-Specific Integrated Circuit.

BLAS Basic Linear Algebra Subprograms.
BLIS BLAS-like Library Instantiation Software.

C-SRAM Computational SRAM.
CACTI Cache Access and Cycle Time model.
CEA Commissariat à l'Énergie Atomique et aux energies alternatives.
COMA Cache-Only Memory Architecture.
CPU Central Processing Unit.
CSR Control-Status Register.
CU Control Unit.
CUDA Computer Unified Device Architecture.

DMA Direct Memory Access.
DMU Data-locality Management Unit.
DSL Domain-Specific Language.
DSM Distributed Shared Memory.

EDVAC Electronic Discrete Variable Automatic Computer.

FANN Fast Artificial Neural Network.

FFTW Fastest Fourier Transform in the West.
FLOPS Floating-Point Operations Per Second.
FPGA Field-Programmable Gate Array.

GCC GNU Compiler Collection.
GPU Graphics Processing Unit.

HPU Host Processing Unit.

IMC In-Memory Computing.
IPC Inter-Process Communication.
ISA Instruction Set Architecture.
ISPC Implicit SPMD Program Compiler.

LAPACK Linear Algebra Package.
LBM Lattice Boltzmann Method.
LFIM Laboratoire de Fonctions Innovantes Mixtes.
LTO Link-Time Optimization.

METEOR Matrix of Elementary Tiles Enabling Optimal Reconfigurability.
MIMD Multiple Instruction, Multiple Data.
MISD Multiple Instruction, Single Data.
MKL Math Kernel Library.
MMIO Memory-Mapped Input/Output.
MPI Message Passing Interface.

NUMA Non-Unified Memory Architecture.

PE Processing Element.
PSE Pattern Stream Encoding.

QEMU Quick EMUlator.

RTL Register Transfer Level.

SCM Storage-Class Memory.
SIMD Single Instruction, Multiple Data.
SISD Single Instruction, Single Data.
SPMD Single Program, Multiple Data.
SRAM Static Random-Access Memory.

UMA Unified Memory Architecture.