Approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA. Application à la qualification des radars et des systèmes d'écoute électromagnétique

Maxime Martelli

NNT : 2019SACLS581

Approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA. Application à la qualification des radars et des systèmes d'écoute électromagnétique

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Traitement du Signal et des Images

Thèse présentée et soutenue à Gif sur Yvette, le 13 décembre 2019, par

MAXIME MARTELLI

Composition du Jury :

Frédéric MAGOULES Professeur, CentraleSupelec (MICS)	Président
Bertrand GRANADO Professeur, Sorbonne Université (LIP6)	Rapporteur
Dominique HOUZET Professeur, Grenoble INP (GIPSA Lab)	Rapporteur
Myriam NOUVEL Responsable Département Simulation, THALES DMS France	Examinatrice
Maxime PELCAT Maître de conférences HDR, INSA (IETR)	Examinateur
Olivier ROMAIN Professeur, Université Cergy Pontoise (ETIS)	Examinateur
Alain MERIGOT Professeur, Université Paris-Sud (SATIE)	Directeur de thèse
Nicolas GAC Maître de conférences, Université Paris-Sud (L2S)	Co-encadrant de thèse
Cyrille ENDERLI Ingénieur de recherche, Thales DMS France	Invité

Thèse de doctorat

Remerciements

Cette thèse a été réalisée conjointement, au sein du Laboratoire des Signaux et Systèmes (L2S - UMR CNRS 8506, École CentraleSupelec, Université Paris Sud) au sein de l'équipe GPI (Groupes Problèmes Inverses), du Laboratoire des Systèmes et Applications des Technologies de l'Information et de l'Energie (SATIE - UMR CNRS 8029, Ecole Normale Supérieure Paris Saclay), ainsi qu'au sein de la Direction Technique de l'entreprise Thales DMS France, sur le site d'Elancourt, sous la forme d'un financement CIFRE.

Mes premiers remerciements s'adressent évidemment à mon directeur de thèse, Monsieur Alain Mérigot, ainsi qu'à Monsieur Jean-François Laneyrie, et Madame Myriam Nouvel, responsables successifs du département Simulation de Thales DMS, pour la confiance qu'ils m'ont accordée pour mener à bien cette thèse, et leur soutien durant ces trois dernières années.

Je tiens ensuite à remercier chaleureusement Messieurs Nicolas Gac et Cyrille Enderli non seulement pour leurs avis et remarques pertinentes au niveau de l'encadrement de mes travaux de thèse, mais aussi par leur proactivité dans l'accompagnement de mes diverses démarches tant administratives que personnelles, ainsi que par leur bienveillance à toute épreuve.

Je saisis également l'occasion pour adresser mes profonds remerciements à Monsieur Bertrand Granado, Professeur à l'université Sorbonne (campus UPMC) au sein du laboratoire LIP6, ainsi qu'à Monsieur Dominique Houzet, Professeur à l'institut Grenoble-INP au sein du laboratoire GIPSA-lab, de l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de cette thèse.

Au sein du L2S, une mention spéciale aux membres des équipes administratives et techniques du laboratoire, qui, malgré la complexité de la gestion d'un laboratoire en parallèle de la fusion de Centrale et de Supélec et de son intégration dans l'Université Paris-Saclay, ont toujours su mener à bien leur accompagnement dans nos différentes missions, mais également dans les tracas quotidiens de la vie de laboratoire.

Au sein de Thales, je tiens également à remercier les collègues que j'ai pu côtoyer, au sein des départements Simulation et Algorithmie, et je tiens à les féliciter d'avoir réussi à survivre à mon humour et à mes jeux de mots invraisemblables pendant si longtemps. Je tiens à remercier l'ensemble des équipes du département simulation de la Direction Technique pour les discussions enrichissantes et les moments de convivialité, et plus particulièrement Timothée pour son aide et son retour d'expérience concernant sa thèse mais aussi pour les nombreux débats sur le sens des choses. Je ne saurais oublier les différents collègues passés par mon OpenSpace, notamment Philippe, Ludovic, Hugo, Guillaume, Paul, Joffrey et Julien qui ont su apporter un vent de fraîcheur pendant ces trois dernières années.

J'aimerais remercier Daniel Charlet, ingénieur de recherche du Laboratoire de l'Accélérateur Linéaire pour m'avoir non seulement permis d'utiliser les ressources matérielles de son laboratoire, mais qui a su porter efficacement les différentes collaborations entre entités de recherche, et promouvoir efficacement la thématique de l'accélération d'algorithmes avec OpenCL sur FPGA.

Je tiens forcément à remercier mes amis, en particulier, Alexis, Thomas, Matthieu, Thuy, Noémi, Fabian, et Maxime, pour les moments passés ensemble, et les longues discussions passionnantes ou moments de détente qui avaient l'avantage de me changer efficacement les idées.

Un remerciement particulier à Adeline, ma compagne, qui a su me soutenir et m'accompagner dans mes périodes d'incertitude, et dont la patience et la bienveillance m'ont poussé à donner le meilleur de moi-même.

Témoins de l'inarrêtable marche du temps, ces trois dernières années ont vu s'éteindre mes grands parents maternels, à qui j'aimerais injustement rendre hommage dans ces lignes, tant la sobriété de celles-ci fait pâle figure face à leur amour, à leur dévouement inconditionnel, et à la grandeur de ce pour quoi ils se sont toujours battus.

Finalement, mes dernières pensées vont à mes frères qui, par la force des choses, ont toujours été là, et dont le soutien tacite est en permanence présent. A mes parents dont la dévotion à notre égard, et l'éducation prodiguée, souvent en contrepartie de sacrifices de leur part, nous a permis à mes frères et moi-même d'arriver là où nous en sommes aujourd'hui.

A ma famille, de sang comme de cœur.

Résumé

A l'heure où l'industrie des semi-conducteurs fait face à des difficultés majeures pour entretenir une croissance en berne, les nouveaux outils de synthèse de haut niveau repositionnent les FPGAs comme une technologie de premier plan pour l'accélération matérielle d'algorithmes face aux clusters à base de CPUs et GPUs.

Mais en l'état, pour un ingénieur logiciel, ces outils ne garantissent pas, sans expertise du matériel sous-jacent, l'utilisation de ces technologies à leur plein potentiel. Cette particularité peut alors constituer un frein à leur démocratisation.

C'est pourquoi nous proposons une méthodologie d'accélération d'algorithmes sur FPGA. Après avoir présenté un modèle d'architecture haut niveau de cette cible, nous détaillons différentes optimisations possibles en OpenCL, pour finalement définir une stratégie d'exploration pertinente pour l'accélération d'algorithmes sur FPGA.

Appliquée sur différents cas d'étude, de la reconstruction tomographique à la modélisation d'un brouillage aéroporté radar, nous évaluons notre méthodologie suivant trois principaux critères de performance : le temps de développement, le temps d'exécution, et l'efficacité énergétique.

Abstract

As the semiconductor industry faces major challenges in sustaining its growth, new High-Level Synthesis tools are repositioning FPGAs as a leading technology for algorithm acceleration in the face of CPU and GPU-based clusters.

But as it stands, for a software engineer, these tools do not guarantee, without expertise of the underlying hardware, that these technologies will be harnessed to their full potential. This can be a game breaker for their democratization.

From this observation, we propose a methodology for algorithm acceleration on FPGAs. After presenting a high-level model of this architecture, we detail possible optimizations in OpenCL, and finally define a relevant exploration strategy for accelerating algorithms on FPGA.

Applied to different case studies, from tomographic reconstruction to the modelling of an airborne radar jammer, we evaluate our methodology according to three main performance criteria : development time, execution time, and energy efficiency.

Acronymes

AAA	Adéquation Algorithme Architecture
ALU	unité logique et arithmétique (*Arithmetic and Logic Unit*)
ASIC	circuit intégré propre à une application (*Application-Specific Integrated Circuit*)
BCLDF	Brouilleur Corrélé Localisé en Distance et en Fréquence
CISC	processeur à jeu d'instruction étendu (*Complex Instruction Set Computer*)
CPU	Processeur Central
CUDA	Common Unified Device Architecture
DAVA	Distance Ambiguë Vitesse Ambiguë
FF	bascule (*Flip-Flop*)
FFT	transformée de Fourier rapide (*Fast Fourier Transform*)
FIFO	premier entré premier sorti (*First In First Out*)
FPGA	circuit logique reprogrammable (*FPGA - Field-Programmable Gate Array*)
GPU	Processeur Graphique (*Graphical Processing Unit*)
HDL	langage de description de matériel (*Hardware Description Language*)
HPC	calcul haute performance (*High-Performance Computing*)
ILP	parallélisme d'instructions (*Instruction-Level Parallelism*)
LUT	table de correspondance (*LUT - Lookup Table*)
NDRK	NDRange Kernel
OpenCL	Open Computing Language
PE	Pipeline Élémentaire

RAM	mémoire vive (*Random Access Memory*)
RISC	processeur à jeu d'instruction réduit (*Reduced Instruction Set Computer*)
RTL	Langage de Transferts de Registres (*Register Transfer Level*)
SaaS	Logiciel en tant que Service (*Software as a Service*)
SEN	Simulateur d'Environnements Numériques
SIMT	Simple instruction multiples threads (*Single Instruction multiple threads*)
SoC	Système sur puce (*System on Chip*)
SWIK	Single Work-Item Kernel
TPU	Processeur neuronaux (*Tensor Processing Unit*)
VLIW	processeur à jeu d'instruction très long (*Very Long Instruction Word*)

Glossaire

Bitstream	Fichier de configuration d'un FPGA. Sous la forme d'une succession de bits, il contient les données de configuration nécessaires pour reprogrammer les connexions et les éléments reprogrammables du FPGA afin d'implémenter les fonctionnalités voulues.
Device	Architecture d'accélération dans la terminologie OpenCL
Kernel	Fonction accélérée dans la terminologie OpenCL
Placement	Etape de positionnement automatique des différentes parties d'un circuit électronique
Routage	Etape de connexion automatique des différentes parties d'un circuit électronique
Taux d'expansion	Désigne le rapport entre le temps simulé et le temps réel dans une simulation (*Une simulation qui, pour 1 seconde de scénario en temps réel met 10 secondes en temps simulé a un taux d'expansion de 10*).

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction

Les performances des architectures informatiques traditionnelles peinent à suivre le rythme de croissance soutenue de la transformation numérique de notre société. À mesure que l'on se rapproche de la limite atomique des transistors, la miniaturisation des circuits électroniques devient de plus en plus complexe, notamment à cause de la difficulté à maîtriser la dissipation énergétique des puces. La recherche de nouveaux relais de croissance pour améliorer la performance des architectures, et l'optimisation logicielle des programmes a motivé notre méthodologie d'approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA, que nous avons appliquée à la reconstruction tomographique et à la qualification des radars et des systèmes d'écoute électromagnétique au sein des laboratoires L2S et SATIE ainsi qu'au sein de la Direction Technique de l'entreprise Thales DMS France.

L'efficacité énergétique et la modularité des architectures Circuits logiques reprogrammables (*FPGAs - Field-Programmable Gate Arrays*), semi-conducteurs constitués de blocs reprogrammables rendent leur utilisation attractive pour de nombreux systèmes embarqués, ou comme solution de prototypage de fonctions spécialisées. Aujourd'hui, bénéficiant des avancées du domaine des semi-conducteurs, ces architectures ont vu croître leurs performances. Leur utilisation dans des serveurs de calculs s'est avérée pertinente comme concurrents aux CPUs. De ces premières constatations émerge l'idée d'utiliser ces architectures pour notre démarche d'Adéquation Algorithme Architecture (AAA), en tant que plateforme susceptible de s'adapter aux spécificités d'une classe d'algorithmes. Nous nous attacherons donc à comparer leurs performances et celles des architectures CPU/GPU pour motiver notre choix.

L'un des principaux freins à une adoption plus large de cette technologie pour l'optimisation des algorithmes est l'apprentissage du savoir-faire nécessaire à leur utilisation. De nombreuses initiatives tant académiques que commerciales ont eu, dès les années 90, pour objectif d'augmenter le niveau d'abstraction de la programmation de ces plateformes, en proposant des langages haut niveau permettant de s'affranchir d'une partie de la complexité de mise en place d'algorithmes sur FPGAs. Plus récemment, les principaux acteurs du marché, Xilinx et Intel, sont allés encore plus loin en proposant des outils basés sur le langage OpenCL, pour l'utilisation des FPGAs comme co-processeur de calculs. Néanmoins, s'il est souvent aisé d'implémenter un algorithme fonctionnel sur FPGA avec ces solutions OpenCL, le doter d'un niveau de performances comparable à celles obtenues avec une description matérielle requiert de la part d'un ingénieur logiciel, un niveau d'expertise quasiment comparable à celle d'un ingénieur matériel. Pour pallier cette difficulté, un des objectifs de notre thèse a été de mettre en place avec le langage OpenCL, des concepts d'optimisation pertinents, pour accélérer les performances d'algo-

rithmes dédiés au traitement du signal.

Un grand nombre de projets industriels, appliqués aux systèmes embarqués intègrent des FPGAs. La conception algorithmique des traitements sur ces supports se fait le plus souvent sur des ordinateurs bureautiques avec des outils comme Matlab, le traitement final étant implémenté ensuite sur les FPGAs à l'aide de langages de description matériel comme le VHDL. Un même algorithme possède alors plusieurs niveaux de représentativités, et si, à l'heure actuelle la rupture est trop importante entre les langages de bas niveau et les langages de haut niveau, l'utilisation du langage OpenCL apporte une interface de modélisation, entre logiciel et matériel, autour d'un socle commun de programmation et c'est dans ce contexte, que dans ce mémoire de thèse, nous avons posé la problématique suivante :

Les nouveaux outils FPGAs de haut niveau basés sur OpenCL permettent-ils une intégration rapide et efficace d'algorithmes dans des projets industriels à architecture complexe ?

Les trois parties de ce mémoire sont structurées comme suit :

Dans la première partie, nous dressons d'abord un constat de l'état actuel de l'industrie des semi-conducteurs, pour ensuite présenter les architectures et les outils nécessaires à la compréhension de nos travaux, en focalisant l'attention sur les architectures FPGAs et le langage OpenCL.

La seconde partie présente le cœur de notre méthodologie OpenCL pour l'accélération matérielle des traitements sur FPGAs. Nous commençons par présenter les métriques correspondantes, pour proposer ensuite une série d'optimisations OpenCL. Dans le chapitre de synthèse, nous présentons notre méthodologie générale d'accélération d'algorithmes en OpenCL sur FPGA que nous incluons dans une démarche d'optimisation multicritères.

Dans la troisième partie, nous appliquons notre méthodologie à des algorithmes de dimension industriels. Pour valider le caractère générique de notre méthode, nous avons choisi un large panel de domaines. Les thématiques abordées sont notamment la reconstruction tomographique en imagerie médicale, la modélisation d'environnements radar et de systèmes d'écoute électromagnétique, ainsi que des algorithmes adaptés de benchmarks d'optimisation. Pour chacune des applications considérées, nous en examinons la faisabilité dans un contexte d'accélération matérielle, sous le prisme de critères tels que la performance brute, le temps d'exécution ou l'efficacité énergétique.

La synthèse des optimisations effectuées permet d'apporter conclusion et perspectives à ces travaux, dans un domaine en constante évolution.

Première partie

État de l’art sur l’accélération des calculs

Chapitre I

Evolution et relais de croissance des technologies à base de semi-conducteurs

Sommaire

I.1 Limites des performances des architectures traditionnelles

Depuis l'apparition des premiers outils jusqu'au développement des communications à distance et des ordinateurs, l'homme a développé théories et techniques pour transférer son intelligence à des mécanismes visant à améliorer son efficacité, et ce faisant, il a repoussé les limites de sa connaissance et étendu son domaine d'action. Mais, chaque avancée théorique se doit d'être déclinée en application réelle, sans quoi elle n'aurait pas d'impact mesurable.

Dans le domaine de l'informatique notamment, alors que les bases de la programmation étaient établies dès le $18^{\text{ème}}$ siècle, il faudra attendre le $20^{\text{ème}}$ siècle pour arriver à une utilisation généralisée de celle-ci. En effet, dès 1725, Basile Bouchon, fils d'un fabricant d'orgues, adapte certains principes de l'horlogerie de l'époque au domaine du tissage, et crée un système d'aiguilles à tisser, programmées par la lecture d'un ruban perforé (Figure I.1a).

(a) Aiguilles de Basile Bouchon (1725)

Source: Maison des Canuts (Lyon)

(b) Machine Jacquard à 400 crochets (1801)

Source: Musée des tissus (Lyon)

FIGURE I.1 – Les premières machines programmables.

Repris et amélioré par Joseph-Marie Jacquard en 1801, ce qui devient la Machine Jacquard (Figure I.1b) est en quelque sorte la plus ancienne machine programmable connue complètement automatique.

En 1834, Charles Babbage eut l'idée de reprendre le principe des cartes perforées du métier Jacquard pour la conception de sa machine à calculer, ou machine analytique [Babbage, Charles, 1851]. Prouesse conceptuelle et technologique pour l'époque, sa contribution majeure fut, en s'intéressant aux méthodes d'automatisation des calculs, de définir les principaux concepts que nous retrouvons dans les ordinateurs actuels. En effet, ses machines avaient des cartes perforées de formats différents suivant qu'il s'agissait de

données ou d'instructions, les unités de contrôles pouvaient faire des sauts conditionnels, et les entrées/sorties étaient séparées. Le premier algorithme était lui inventé par une femme, Ada Lovelace [Cellania, 2015] (qui donnera son nom au langage de programmation ADA). Première programmeuse de l'histoire, elle explique que la machine au-delà des traitements numériques sera amenée à traiter des symboles et à effectuer un traitement analytique. Mais il faudra attendre un siècle pour que la découverte technologique des transistors, véritable clef de voûte du matériel informatique actuel, permette l'évolution des architectures informatiques.

En décembre 1947, John Bardeen, William Shockley, et Walter Brattain inventent le premier transistor [Bardeen et al., 1948], devançant de peu les chercheurs Herbert Mataré et Heinrich Welker de la Compagnie des Freins et Signaux à Paris, qui présentent leur version indépendante du transistor [Herbert Mataré and Heinrich Welker, 1948] en juin 1948. Depuis lors, les évolutions technologiques et théoriques se succéderont à un rythme soutenu, pour nous amener en un quart de siècle, avec l'évolution conjointe des communications à distance, aux ordinateurs actuels.

En s'intéressant à l'évolution du coût des puces électroniques, Gordon Earle Moore, co-fondateur d'Intel, énonce en 1965 dans un article publié dans le journal *Electronics* [Gordon Earle Moore, 1965] ce qui est désormais connu sous le nom de la première loi de Moore :

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer." Moore, 1965

Ce premier énoncé part du constat que la complexité entre 1959 et 1965 des semi-conducteurs a doublé chaque année à coût constant, et de cette observation, il postule la poursuite empirique de cette évolution.

En considérant le nombre de transistors par circuits intégrés (1 en 1959, 64 en 1965), nous vérifions bien le doublement (I.1) tous les ans de la complexité des circuits intégrés. En extrapolant à l'aide de cette même formule le nombre de transistors que l'on pouvait espérer avoir en 1975 (I.2), nous obtenons 65536 transistors par circuits intégrés, ce qui est cohérent avec l'énoncé précédent de la loi de Moore.

$$2^{(1965-1959)} * 1 = 64 \tag{I.1}$$

$$2^{(1975-1965)} * 64 = 65536 \approx 65000 \tag{I.2}$$

Dans l'intégralité de son article, il s'intéresse principalement à l'équilibre, lors de la miniaturisation des composants, entre gains (tant en termes de performance que de consommation) et coûts : il est surtout question de la rentabilité des semi-conducteurs, d'un point de vue industriel.

En 1975, Gordon E. Moore reprend sa première loi, et l'applique au nombre de transistors par microprocesseurs (et non plus par simples circuits intégrés), et il en modifie le ratio d'évolution : le nombre de transistors sur une puce de silicium est alors censé doubler tous les deux ans. C'est cet énoncé corrigé qui est retenu comme la deuxième loi de Moore.

Ces lois, qui sont les seules réellement énoncées par Gordon E. Moore, sont empiriques, et la seconde a été, durant toutes ces années, une précieuse feuille de route pour la recherche et le développement dans le domaine des semi-conducteurs. L'objectif principal était de permettre aux industriels de définir et de maîtriser le modèle économique du domaine en autolimitant les évolutions technologiques à l'aide de la loi de Moore.

Depuis la fin des années 1970, les architectures informatiques ont évolué de façon exponentielle jusqu'au début des années 2000. Cette croissance, illustrée à la Figure I.2, subit aujourd'hui un effet de seuil. Nous allons examiner ci dessous dans le détail, cette évolution.

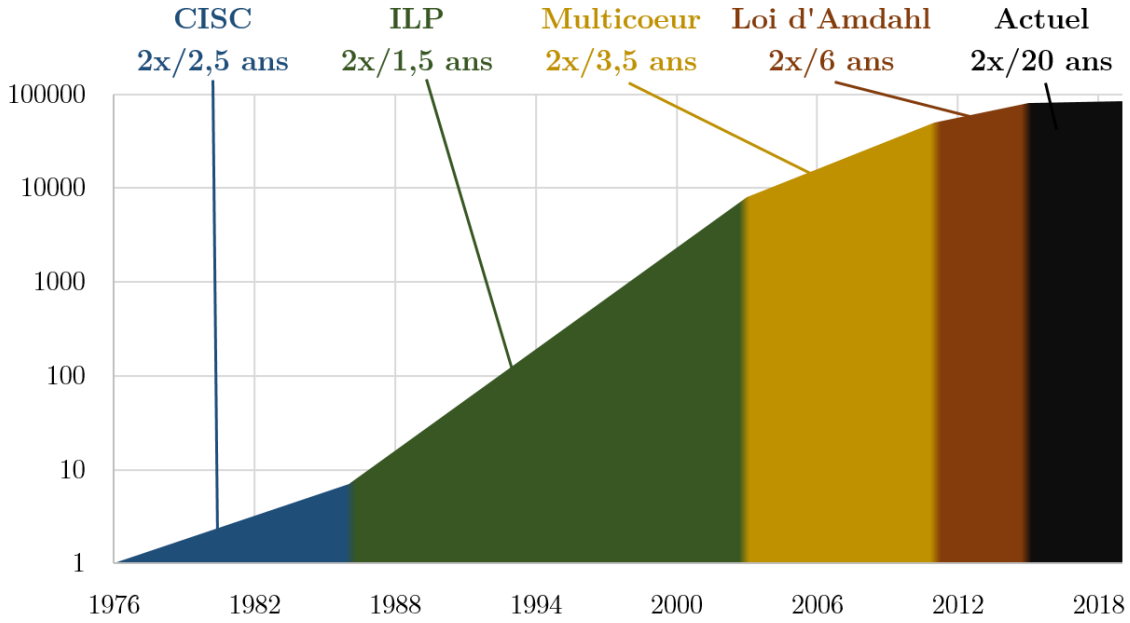


FIGURE I.2 – Evolution de la performance moyenne des ordinateurs (par rapport au VAX11-780) mesuré sur la suite de benchmark SPECint [Dixit, 1993]

Source: Computer Architecture, *a Quantitative Approach* [Patterson and Hennessy, 2007]

Dans les années 90, la promesse d'une croissance régulière des performances moyennes des semi-conducteurs a entraîné l'utilisation de mémoires de taille toujours plus importante, et a impliqué la complexification des jeux d'instructions. Simultanément, portés par l'explosion du nombre de transistors disponibles grâce à la miniaturisation, les contrôleurs d'instructions devenaient de plus en plus complexes, exploitant toujours plus le parallélisme d'instructions (*Instruction-Level Parallelism*) (ILP) pour augmenter la performance des architectures.

Mais, la principale force du parallélisme d'instruction en est aussi son principal incon-

vénient. En effet, la performance du parallélisme d'instructions s'appuie sur différents mécanismes, comme l'ordonnancement dynamique [Tomasulo, 1967], les instructions vectorielles, l'exécution multi-flot, ou encore la prédiction de branchement [Smith, 1981], [Jimenez and Lin, 2001], [Seznec, 2006]. Ce dernier consiste à tenter de prédire le résultat d'un branchement, ce qui a pour conséquence de rendre le pipeline d'instructions plus efficace en cas de réussite, mais rajoute un surcoût, tant au niveau de la consommation, qu'au niveau des performances en cas d'échec. Aussi, augmenter la taille d'un pipeline d'instructions entraîne une difficulté croissante à garantir un taux d'échec faible sur les prédictions de branchement, ce qui diminue l'efficacité générale d'une architecture donnée. À partir de la suite de benchmark SPECint, des chercheurs ont quantifié, pour une série d'algorithmes [Limaye and Adegbija, 2018], qu'un Processeur Central (CPU) Intel Xeon E5-2650L avait en moyenne 19% d'instructions gâchées à cause de prédictions erronées.

En plus de la limitation du parallélisme d'instructions vers le milieu des années 2000, une autre difficulté est venue ralentir l'augmentation des performances des semi-conducteurs : la fin de la règle de Dennard. En 1974, Robert H. Dennard observe que la tension et le courant électrique sont proportionnels à la dimension linéaire d'un transistor [Bohr, 2007]. Ainsi, à mesure que la taille des transistors diminue, ces valeurs baissent, et il en extrapole que le rapport entre la densité et la puissance reste constant à mesure de l'évolution de la miniaturisation. Cet énoncé, mis en parallèle avec la loi de Moore, semblait permettre une évolution rapide à coût et consommation électrique constante des architectures informatiques [Frank et al., 2001]. Mais, depuis 2004, les difficultés à maîtriser la dissipation thermique des architectures ont donné un coup d'arrêt à cette règle, qui n'est dorénavant plus d'actualité.

En conséquence, les contraintes technologiques des architectures informatiques, liées à la baisse de performances du parallélisme d'instruction, ont conduit les industriels à s'intéresser aux architectures multi-coeurs.

Si nous examinons ces architectures du point de vue des performances, leur croissance a été prolifique dans les années 2000, mais il ne faut pas oublier qu'augmenter le nombre de cœurs d'une architecture n'augmente pas de manière proportionnelle le parallélisme correspondant. La corrélation entre ces deux facteurs est donné par la Loi d'Amdhal [Mark D. Hill and Michael R. Marty, 2008], et souligne le fait qu'un programme est limité par sa partie séquentielle, quelle que soit l'architecture sous-jacente.

En observant l'allure générale de l'évolution des performances présentée à la Figure I.2, l'accumulation des obstacles énoncés précédemment a comme conséquence directe un essoufflement de la croissance du secteur des semi-conducteurs.

Si nous examinons les performances obtenues dans ce domaine, suivre la cadence énoncée par la loi de Moore s'est avéré ces dernières années complexe, voire impossible comme nous allons l'illustrer avec l'exemple d'Intel. L'entreprise avait annoncé en 2013 pouvoir produire des puces de 10nm de finesse de gravure pour 2015, avant de repousser ce délai à 2016, puis à fin 2017. En décembre 2018, Dr. Venkata Renduchintala, PDG d'Intel Corporation, annonçait que leurs recherches concernant les processeurs 10 nm avaient dû être revues de fond en comble, mais que la nouvelle direction donnée au projet était désormais assurée. Finalement, début 2019, au CES2019 Las Vegas, Intel a annoncé la disponibilité des puces en 10 nm pour la fin d'année 2019. Ce délai pourra-t-il être respecté, ces retards successifs ne seraient-ils pas justement le symbole des

difficultés à accroître les performances du domaine ?

Même si l'entreprise a toujours eu une avance considérable sur ses concurrents dans la production de puces, cet avantage est en train de diminuer rapidement. En effet, à densité égale, Intel reste plus performant que ses principaux compétiteurs (TSMC et Samsung) (Figure I.3).

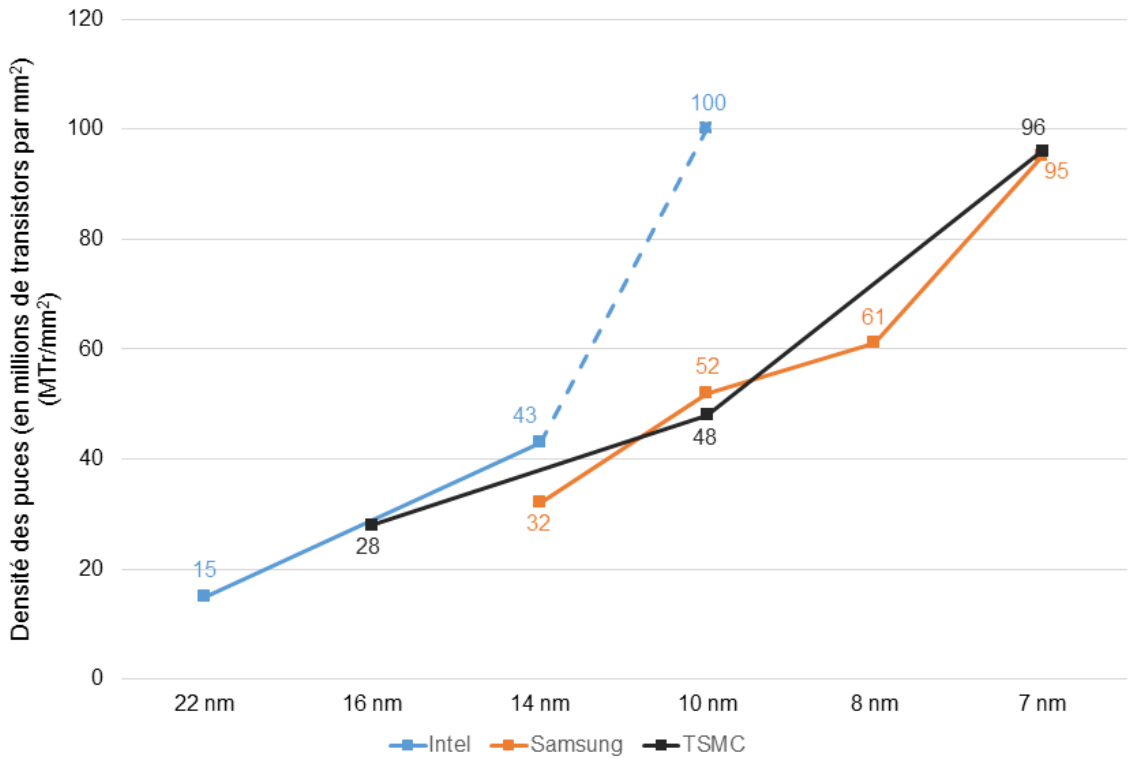


FIGURE I.3 – Comparaison de la densité de différentes puces électroniques.

Mais ces derniers ont déjà réussi à graver des puces en 10 nm, et arrivent même aujourd'hui à produire des puces mobiles en 7 nm, voire en 5 nm. Le danger pour Intel est donc d'avoir des puces rivales qui, en termes de densité et de performances, soient équivalentes voire meilleures que ses propres puces, bloquées à 14 nm de finesse de gravure et dont l'entreprise n'arrive plus à tirer de gain de performances significatif.

Cette difficulté à miniaturiser les puces de la part du plus grand fabricant mondial de semi-conducteurs est un indicateur puissant du problème auquel est confronté le domaine dans sa globalité. Certes, la miniaturisation permet d'augmenter les performances des puces, de réduire leur coût unitaire, et également d'amoindrir le délai entre les fils et les transistors ainsi que la consommation locale d'énergie. Mais, la nécessité de réduire la tension d'alimentation des transistors pour mieux contrôler la dissipation d'énergie implique que les charges de ces derniers sont plus facilement perdues. À cela s'ajoute qu'au fur et à mesure, les composants deviennent de plus en plus fragiles. Aussi, une augmentation de la densité des semi-conducteurs implique d'augmenter le ratio des transistors utilisés pour la redondance des informations, ce qui en réduit d'autant le gain en performances.

Le constat est donc clair : en approchant de la limite physique possible de la taille des transistors, la miniaturisation des architectures traditionnelles n'est plus viable sur le long terme. Aussi, l'Association des Industriels des Semi-conducteurs (SIA), composée des grands industriels du domaine (comme AMD, IBM, Intel, ARM, Qualcomm, Xilinx, Texas Instruments, . . .) a annoncé la fin effective de la loi de Moore pour 2021 [Iec, 2015]. Cette annonce clôture cinquante-six années d'évolution qui ont vu les puces en silicium passer de 2300 transistors par microprocesseur (Intel 4004 [Aspray, 1997]) en 1971 à plus de 19 milliards de transistors en 2017 pour le SoC d'AMD Epyc intégrant 32 cœurs.

En conséquence, il apparaît clairement que l'amélioration des performances par la miniaturisation des architectures traditionnelles est limitée. Pourtant, il existe de nombreux axes de croissance possibles pour cette industrie que nous allons évoquer ci-après.

I.2 Relais de croissance de l'industrie des semi-conducteurs

Dans le domaine des composants, les fondeurs poursuivent toujours leurs recherches pour miniaturiser les transistors jusqu'à leur taille limite, en s'intéressant par exemple à des procédés de lithographie novateurs comme l'Extreme Ultra Violet [Christian Wagner and Noreen Harned, 2010], mais l'inéluctable barrière atomique contraint les acteurs du domaine à trouver d'autres relais de croissance pour pousser les performances des architectures informatiques toujours plus loin.

C'est pourquoi nous assistons également à l'émergence de nouveaux concepts d'ordinateurs, comme les ordinateurs optiques et quantiques. Ces thématiques très présentes dans les communautés scientifiques commencent à porter leurs fruits, notamment par la démonstration de prototypes fonctionnels et prometteurs [DeBenedictis et al., 2018], à l'image d'IBM qui a dévoilé au CES2019 le premier système d'ordinateur quantique à usage scientifique et commercial [IBM, 2019].

Dans le cas des ordinateurs quantiques, leur principe repose sur l'exploitation du comportement probabiliste des atomes, sortant ainsi de la physique traditionnelle pour aborder le domaine de la physique quantique. Cette transformation laisse entrevoir un certain nombre d'avantages, comme une précision plus importante des calculs, ou encore la possibilité d'avoir des simulations physiques à l'échelle atomique. Néanmoins, cette puissance promise amène également son lot de problématiques. En effet, ces ordinateurs seront en théorie capables d'effectuer des factorisations entières en un temps polynomial au lieu d'un temps exponentiel pour les ordinateurs non quantiques, ce qui aura comme conséquence directe de casser certaines techniques de cryptage traditionnelles comme le RSA, qui est notamment utilisé pour les certificats SSL. Il a été démontré [Craig Gidney and Martin Ekerå, 2019], qu'il ne faudrait que 8 heures à un ordinateur quantique avec un nombre suffisant de qubits pour venir à bout d'un chiffrement RSA 2048 bits, au lieu d'une vingtaine d'années actuellement en utilisant les architectures classiques.

Ces résultats théoriques, bien qu'aujourd'hui limités par la capacité à gérer un certain nombre de qubits en parallèle, montrent bien que toute solution technologique se doit d'être suffisamment quantifiée afin de préparer en amont les transformations majeures du domaine.

Un autre relais de croissance consiste à faire évoluer la géométrie des puces, en s'émancipant des puces en deux dimensions par l'exploration de l'empilement 3D des

transistors. Bien que les progrès dans cet axe technologique aient été relativement lents, notamment à cause de la surchauffe induite par l'empilement des modules, Intel a révélé fin 2018 l'architecture Foveros

Une des applications pratiques de l'exploitation de cette verticalité est de pouvoir créer des architectures hybrides avec des cœurs basse consommation (gamme des CPUs Atom) d'un côté et des cœurs haute performance (gamme des CPUs Core), avec des composants pouvant communiquer plus rapidement et avec un meilleur débit entre eux. La Figure I.4 illustre ce genre de puces, qu'Intel compte commercialiser en fin d'année 2019.

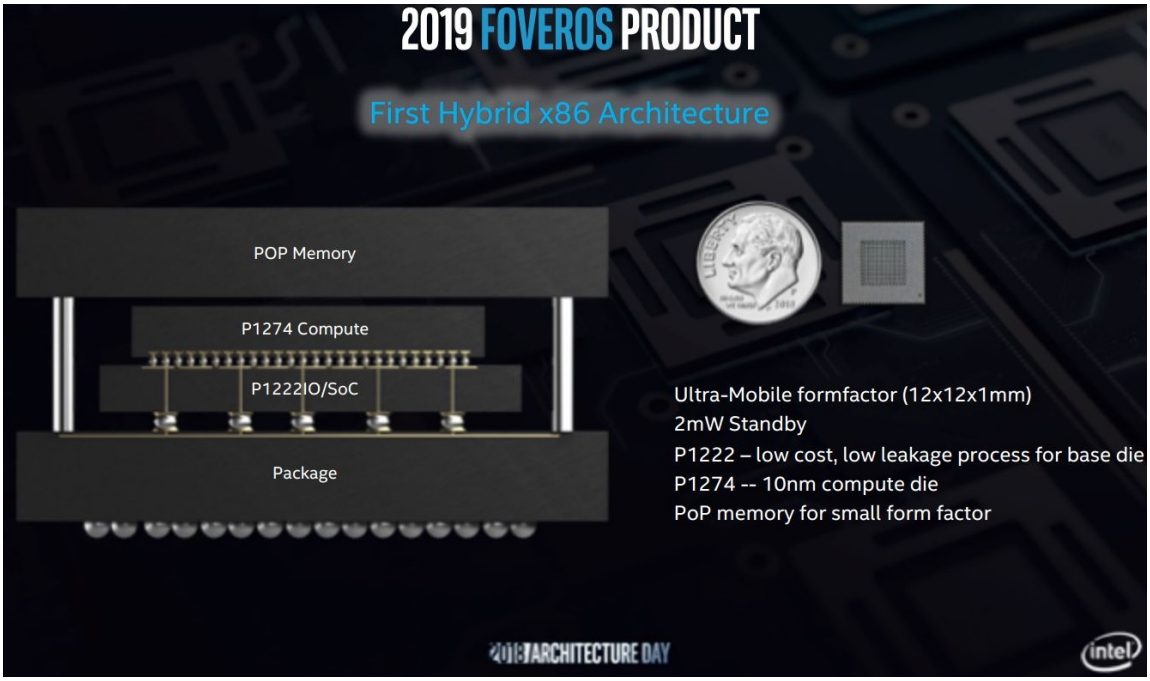


FIGURE I.4 – Intégration 3D, architecture hybride Atom-Core

Source: Intel Architecture Day 2018

La solution d'Intel de passer par la troisième dimension comme relais de croissance desservira majoritairement la conception de puces à base d'architectures hétérogènes à l'image de la puce illustrée à la Figure I.4. Cette tendance à repenser les architectures traditionnelles telles que nous les connaissons est en réalité une nécessité plus qu'une piste de recherche. En effet, il s'agit dorénavant non pas de s'intéresser à l'évolution de la performance brute des architectures, mais bien à leurs performances effectives, ce qui passe par une démarche d'adéquation algorithme architecture.

Concernant les performances réelles des architectures informatiques traditionnelles, un paradoxe apparaît. Au début de l'ère des ordinateurs personnels, les processeurs étaient lents par rapport à ceux d'aujourd'hui, et il était alors nécessaire de passer une portion significative du temps de développement à l'optimisation des programmes. Mais, au fur et à mesure que les architectures gagnaient en puissance, les fonctionnalités proposées se sont étoffées, et la contrainte d'optimisation des programmes a peu à peu

diminué, jusqu'à un stade où la majorité du temps de développement était consacré à l'implémentation rapide de nouvelles fonctionnalités, en ne laissant que peu de temps pour l'optimisation. Le constat est alors flagrant. L'impression de lourdeur des programmes semble croître plus rapidement que les performances des architectures. Ce phénomène, déjà formulé en 1995 par Niklaus Wirth sous le nom de la loi de Wirth [Niklaus Wirth, 1995] démontre bien que la course effrénée aux nombres de lignes de codes écrites à défaut de leur optimisation est pénalisante, notamment parce que les utilisateurs finaux des technologies grand public n'y retrouvent pas le gain en performance des architectures matérielles.

De plus, rien ne sert d'augmenter la vitesse d'un processeur si le reste des composants d'un ordinateur classique n'est pas dimensionné en adéquation. Aussi, un utilisateur sera bien plus satisfait d'un changement de son disque dur à plateau pour un disque SSD, ou encore d'une augmentation de sa mémoire vive (RAM), que d'un changement de processeur.

Aussi, dimensionner nos plateformes informatiques, tant au niveau des composants macroscopiques qu'au niveau des architectures internes, est désormais crucial dans une ère où l'évolution des performances brutes devient de plus en plus limitée, et où il s'agit dorénavant de s'intéresser à l'amélioration des performances effectives des architectures.

Cette observation fait émerger un autre relais de croissance, qui passe par une meilleure optimisation logicielle des programmes, en adéquation avec les architectures cibles, ce qui représente une partie de ce que l'on appelle la démarche d'Adéquation Algorithme Architecture (AAA).

I.3 Hétérogénéité des architectures traditionnelles et adéquation algorithme architecture

Des remarques précédentes, il découle que les avancées technologiques traditionnelles sont vouées à atteindre un palier de performances difficilement franchissable, d'où l'idée d'utiliser et de concevoir des architectures qui s'adaptent aux spécificités d'une classe d'algorithmes, au lieu d'utiliser une architecture polyvalente comme un CPU pour effectuer toutes les tâches d'un programme donné. Cette démarche d'adéquation algorithme architecture, qui peut bien sûr être implémentée en parallèle de n'importe quel autre relais de croissance, est donc la solution la plus logique à adopter.

Impulsée par le domaine de la téléphonie mobile, nous assistons à la démocratisation des puces hétérogènes. Par exemple, la puce A11 d'Apple [Dilger, 2017], présente notamment dans les iPhones 8, 8 Plus, et X, intègre de nombreux composants spécialisés (Figure I.5). Elle est constituée d'un CPU ARM à 6 cœurs [1], d'un Processeur Graphique (*Graphical Processing Unit*) (GPU), mais également d'un processeur pour le traitement d'images, tels que les estimations de luminosité par exemple, ou encore l'atténuation du grain en temps réel sur les photos par exemple. En outre, la puce A11 inclut également un composant spécifique pour les réseaux de neurones, appelé « Neural Engine », qui permet d'effectuer de très nombreuses tâches liées aux réseaux de neurones (détection

1. Dont deux cœurs haute performance, et quatre cœurs basse consommation.

faciale, reconnaissance de motifs et classification d'images, ...) avec une efficacité énergétique supérieure aux GPUs et aux CPUs.

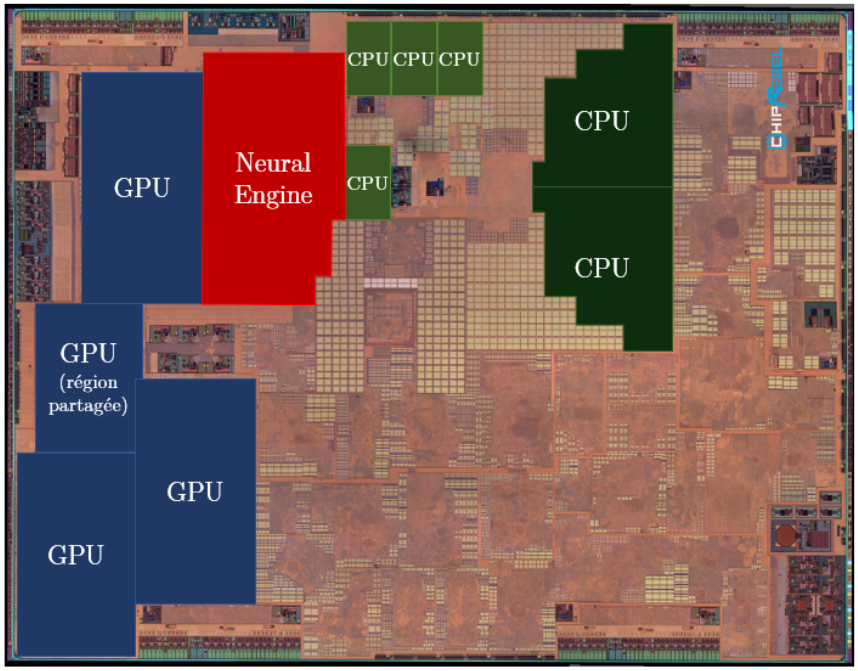


FIGURE I.5 – Puce A11 Bionic d'Apple (2017)

Source: ChipRebel

Mais sans forcément s'intéresser au domaine du mobile qui a toujours su gérer les contraintes fortes tant au niveau de la consommation que des performances, le constat est que l'intensité arithmétique des calculs à effectuer augmente toujours plus vite que les performances brutes des architectures. En effet, nous trouverons toujours des cas d'applications qui nécessitent une puissance de calcul plus importante que les architectures actuelles, et ce quelque soit la puissance de la machine à notre disposition.

C'est pourquoi cette tendance à spécialiser certains traitements, à avoir plusieurs coprocesseurs spécialisés au sein d'une même puce en silicium ou d'une architecture de calcul, est désormais indispensable pour repousser les limites technologiques. Ce tournant, reposant sur le constat déjà évoqué à la section I.1, se retrouve non seulement dans le domaine des systèmes embarqués, mais également dans le domaine du serveur et de l'informatique générale, avec l'intégration de technologies reprogrammables dans les clouds Azure ou Amazon.

L'avènement ces dernières années de nouvelles problématiques, comme la nécessité de limiter la consommation énergétique dans les serveurs, a suscité un regain d'intérêt pour les architectures reprogrammables dans l'industrie de l'informatique de masse, et notamment pour les FPGAs. Très largement utilisés dans des domaines spécifiques comme les systèmes embarqués [Garcia et al., 2006] et les systèmes critiques [Wegrzyn, 2001], leur adoption dans les centres de données est en pleine expansion notamment au sein d'Amazon Web Services ou encore du cloud Azure de Microsoft. Cette adoption

rapide s'explique ainsi : en plus de pouvoir être reprogrammées efficacement, ces architectures permettent une maîtrise de la consommation énergétique inégalée par rapport aux architectures concurrentes conventionnelles comme les CPUs et les GPUs.

Aussi, les analystes du domaine voient le domaine des serveurs et des centres de données comme le premier marché pour les FPGAs pour les décennies à venir, et, fort de ce constat, Intel s'est porté acquéreur d'Altera le 28 décembre 2015 afin de consolider sa place dans ce marché à très fort potentiel.

L'élargissement du marché des FPGAs, que ce soit par leur intégration dans des serveurs, pour des calculs en co-processing, ou par leur utilisation pour l'informatique embarquée, doit s'accompagner d'une simplification des outils, afin de les rendre accessibles au plus grand nombre et augmenter significativement la proportion d'utilisateurs pouvant prendre en main cette technologie. C'est pourquoi les deux principaux industriels du marché des FPGAs, Xilinx et Intel, ont commencé à repousser les limites du niveau d'abstraction nécessaire pour programmer un FPGA, en allant au-delà des outils de programmation qui étaient jusqu'alors disponibles. Ainsi, il a pu être constaté une forte émergence de nouveaux ateliers de développement résolument axés vers la programmation des FPGAs au niveau logiciel et non plus matériel, poussés par cette forte demande d'abstraction de la part des utilisateurs, et par l'opportunité d'un marché en nette augmentation pour les constructeurs de FPGAs.

L'évaluation de ces nouveaux outils, en parallèle de l'étude de leur utilisation dans des projets d'envergure est l'axe majeur de cette thèse.

Chapitre II

Architectures CPU/GPU/FPGA

Sommaire

La complexité croissante des problèmes traités a favorisé l'émergence d'architectures dédiées supportées par des langages qui leur sont adaptés. Après avoir rappelé les concepts de Langage dédié (Domain Specific Language - DSL), d'Architecture dédiée (Domain Specific Architecture - DSA), et brièvement présenté les principaux types de semi-conducteurs, nous détaillons les architectures hétérogènes CPU/GPU/FPGA et mettons en exergue les mécanismes spécifiques qui leur permettent d'exprimer différents types de parallélisme. Dans la dernière section, nous présentons les principes généraux du standard de programmation OpenCL qui serviront à comprendre certaines notions abordées dans la Partie 2.

II.1 Architectures et langages dédiés

Langage dédié (Domain Specific Language - DSL)

Un DSL désigne tout langage de programmation qui a pour but de répondre aux spécificités d'un domaine précis, en opposition à un langage comme le C ou le C++ qui est indépendant d'un domaine particulier. Nous pouvons citer par exemple le HTML pour les pages internet, certains outils [1] de Matlab pour la programmation matricielle, ou encore Maple ou Mathematica pour les mathématiques symboliques. La force de ces langages se trouve dans leur symbiose avec l'écosystème général. En effet, l'ensemble des formules d'un tableur Excel est considéré comme un DSL, en cela qu'il permet la manipulation des bases de données et des analyses statistiques, ce qui représente un domaine d'application spécifique.

L'avantage majeur d'un DSL est d'apporter un gain de temps lors de la programmation de nouvelles applications dans un même domaine. En effet, ce langage étant créé sur mesure pour celui-ci, sa réutilisation en est facilitée, le plus souvent grâce à des librairies optimisées. Néanmoins, sa principale difficulté réside dans sa programmation, ou plus précisément dans l'apprentissage du langage, qui peut s'avérer longue si la documentation n'est pas assez étoffée.

Architecture dédiée (Domain Specific Architecture - DSA)

Le tenant matériel d'un DSL est l'architecture dédiée. Il s'agit d'architectures adaptées aux traitements courants d'un domaine particulier, comme par exemple les puces graphiques, dont l'émergence à la fin des années 90 s'explique par la nécessité de pouvoir manipuler les flux d'images efficacement.

De nos jours, de très nombreuses architectures spécialisées ont vu le jour, à l'image des processeurs neuronaux (Tensor Processing Units) qui sont des processeurs optimisés pour les calculs inhérents aux réseaux de neurones.

Comme présenté en Section I.3, du fait des limites de performance du domaine des semi-conducteurs, la démarche d'adéquation algorithme architecture s'est imposée comme un relais de croissance, ce qui a induit l'émergence de nouvelles architectures non seulement capables d'exécuter un type d'algorithme efficacement, mais également de fonctionner de concert avec d'autres architectures.

1. Toolbox

Aussi, quand nous considérons un projet d'envergure, il est rare qu'une seule architecture puisse être efficace sur tous les traitements à implémenter, et la première étape consiste souvent à segmenter correctement les fonctionnalités du projet afin de dimensionner la plateforme hétérogène finale.

Il convient de préciser qu'une architecture peut être dédiée à plusieurs domaines. Ainsi, les GPUs sont efficaces non seulement sur la gestion des flux graphiques, mais également pour les réseaux de neurones et les calculs parallèles.

L'intérêt de concevoir une DSA permet de standardiser facilement le domaine en s'affranchissant des fonctionnalités inutiles des architectures générales telles que les CPUs.

II.2 Architectures des circuits logiques

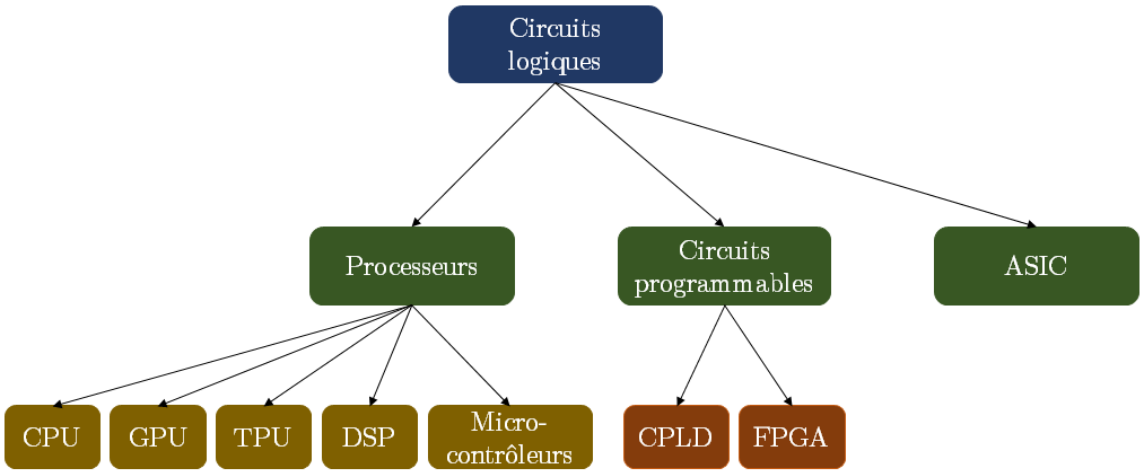


FIGURE II.1 – Hiérarchie des architectures usuelles à base de circuits logiques

Avant de rentrer dans le détail de certaines architectures d'intérêt (notamment les CPUs, GPUs, et FPGAs), commençons par présenter les catégories usuelles de circuits logiques, illustrées en Figure II.1.

Processeurs

La catégorie des processeurs regroupe les circuits électroniques qui interprètent et exécutent des instructions. Nous retrouvons, entre autres, les architectures suivantes :

— CPU : exécute les instructions présentes dans un langage informatique, et inclut entre autres des registres, une unité de contrôle pour gérer les instructions et les entrées sorties, ou encore une unité de calcul. Son jeu d'instruction est très vaste, ce qui le rend polyvalent pour de nombreux types de calculs.

— GPU : processeur spécialisé pour les besoins de l'affichage graphique, ces architectures sont conçues pour manipuler très efficacement des données larges avec un jeu d'instruction réduit. Leur structure s'appuie sur un parallélisme de données massif.

— TPU/NPU : d'une conception récente, ce sont des processeurs adaptés au besoin en calcul des réseaux de neurones.

— DSP : architecture spécialisée pour le traitement du signal, permet d'effectuer des manipulations comme des filtrages ou des compressions de signaux analogiques. S'ils ont souvent une bonne efficacité énergétique, leur performance brute est toutefois limitée.

Circuits programmables

Les circuits programmables, quant à eux, ont la particularité de pouvoir être reconfigurés à souhait après fabrication, grâce à une cascade de cellules logiques connectables. Ce degré de liberté dans la configuration des connexions entre ces cellules permet de réaliser des fonctions numériques complexes à partir de ces blocs de base, et ainsi d'optimiser le flux de données numériques à chaque algorithme. Dans cette catégorie, il est possible de différencier deux types d'architectures.

— CPLD : de conception plus ancienne, les fonctions internes sont regroupées en macro-cellules composées de portes logiques. Le routage fixe permet une fréquence de fonctionnement élevée et indépendante de l'algorithme implémenté. Parce que les données sont dans des structures mémoires non volatiles, ces architectures sont plus sécurisées que les FPGAs qui utilisent à l'inverse de la mémoire volatile.

— FPGA : comprenant un très grand nombre de blocs logiques et d'interconnexions, le temps de génération d'un algorithme sur ces architectures est assez long, mais leurs performances théoriques dépassent largement les CPLDs.

Si les CPLDs permettent un prototypage très rapide de fonctions simples, il est tout de même préférable, pour la majorité des cas d'utilisation de logique reprogrammable, d'utiliser les FPGAs modernes.

ASIC

La notion d'circuit intégré propre à une application (*Application-Specific Integrated Circuit*) (ASIC) est souvent ambigüe. En effet, au sens strict du terme, un CPU est un ASIC. Dans cette section, nous considérons qu'un ASIC est une architecture adaptée à une application précise. Il s'agit donc de la solution d'architecture la plus optimale en termes de performance et de consommation. Non reprogrammable, la puce ASIC est donc peu évolutive, et cette solution est utilisée lors de la phase de commercialisation d'un produit pour des volumes importants.

Nous allons dans les prochaines sections présenter plus en détails les trois architectures d'intérêt dans nos travaux, à savoir les CPUs, les GPUs, et les FPGAs.

II.3 CPU : architecture de référence

II.3.1 Principe

Le fonctionnement de cette architecture, qui est présente dans la grande majorité des ordinateurs actuels, peut se subdiviser en trois parties :

— le chemin de données

— l'unité de contrôle

— le jeu d'instruction

Chemin de données

Le chemin de données est la partie matérielle du processeur qui stocke et effectue les opérations. Elle est composée principalement d'une unité logique et arithmétique (*Arithmetic and Logic Unit*) (ALU) où s'exécutent les calculs, ainsi que de deux structures mémoires : les registres pour la mémoire temporaire rapide, et la RAM pour le stockage des données et des programmes en cours d'exécution.

L'ALU, quant à elle, s'occupe des calculs, et est composée de circuits électroniques qui permettent de manipuler les données en réalisant principalement les fonctions suivantes :

— stockage des données : certaines données calculées ou récupérées d'autres éléments du circuit peuvent y être stockées temporairement, et l'ALU sert alors de régulateur,

— calculs arithmétiques : on retrouve entre autres les opérations classiques binaires, comme l'addition, la soustraction, la multiplication, et la division, mais aussi les comparaisons (tests d'égalités/d'inégalités, ...),

— opérations logiques : par exemple les ET, OU, NON.

Il existe de nombreux types d'ALU, certaines permettant de manipuler différentes précisions de nombres (virgule flottante en simple/double précision) là où d'autres permettent d'effectuer des calculs vectoriels ou mathématiques plus avancés, comme les logarithmes, ou encore les fonctions trigonométriques.

Unité de contrôle

Elle permet de faire le lien entre les programmes et le chemin de données. En effet, elle est chargée de traduire les instructions du programme en signaux de contrôle, puis d'exécuter ces instructions dans le bon ordre. Comme cette unité est cadencée par la fréquence du processeur, la valeur de celle-ci sera donc directement corrélée à la vitesse de traitement d'un programme.

De manière plus précise, une unité de contrôle se décompose en quatre composants importants :

— le pointeur ordinal (*PC, Program counter*), qui est un registre contenant l'adresse mémoire de l'instruction en cours d'exécution [2],

— la mémoire d'instruction, qui contient les différentes instructions d'un programme,

— l'unité de décodage d'instructions, qui, à partir du langage machine, produit les signaux corrects pour le chemin de données, ce qui permet à l'ALU de savoir quelles opérations effectuer.

— l'unité de contrôle des branchements, qui détermine la prochaine valeur du pointeur ordinal.

2. Sur certaines architectures, le PC stocke l'adresse mémoire de la prochaine instruction à exécuter.

Jeu d'instruction

Le jeu d'instruction représente l'ensemble des instructions qu'un processeur peut comprendre et donc exécuter. Ce sont elles qui permettent de configurer les différents éléments présentés ci-dessus, et un programme tournant sur CPU ne consiste en fait qu'à l'interprétation d'une suite d'instructions du jeu d'instruction correspondant.

Depuis les années 1980, de nombreuses familles de processeurs ont vu le jour, les deux principales étant le processeur à jeu d'instruction étendu (*Complex Instruction Set Computer*) (CISC), et le processeur à jeu d'instruction réduit (*Reduced Instruction Set Computer*) (RISC). Il en existe également d'autres familles, comme les processeurs vectoriels, les architectures en flot de données, ou encore les processeurs basés sur le processeur à jeu d'instruction très long (*Very Long Instruction Word*) (VLIW), mais nous ne les détaillerons pas ici.

Un processeur CISC peut, comme son nom l'indique, effectuer un grand nombre d'opérations complexes, qui peuvent prendre un certain nombre de cycles d'horloge du processeur. Un processeur RISC quant à lui, a un jeu d'instruction réduit, et chaque instruction est effectuée en moyenne en un seul cycle d'horloge.

Les différences majeures entre ces deux types de parallélisme se situent au niveau de la vitesse d'exécution d'une instruction et de la compilation des programmes.

En effet, le jeu d'instruction CISC nécessite moins de compilation, puisque la plupart des opérations classiques peuvent se traduire en une instruction (par exemple la fonction puissance), là où un processeur RISC devra découper en une suite d'instructions élémentaires ces opérations. Par contre, même si la vitesse d'exécution peut sembler similaire pour ces deux types de jeu d'instruction, la simplicité de l'architecture RISC permet d'intégrer plus de registres ce qui la rend plus performante qu'une architecture CISC.

Aussi, un processeur CISC est plus polyvalent qu'un RISC qui demande une compilation plus complexe, mais la régularité et la simplicité des instructions de ce dernier type permettent un pipeline plus efficace ainsi que de meilleures performances notamment grâce à un nombre de registres plus importants.

La Figure II.2 décrit le fonctionnement simplifié d'un CPU.

À partir d'un programme exprimé en langage machine, le contenu du compteur du compteur ordinal est écrit sur le bus d'adressage, ce qui permet de récupérer l'instruction correspondante en code machine après écriture du signal correspondant sur le bus de contrôle. Cette instruction, récupérée dans le registre d'instruction, est ensuite décodée, et différents signaux sont générés pour paramétrer son exécution, qui est prise en charge par l'ALU, avec éventuellement gestion des branchements par l'unité de contrôle. Finalement, le compteur ordinal est incrémenté et la prochaine instruction peut être traitée.

Ce traitement circulaire se faisait de manière séquentielle sur les premiers CPUs, mais il existe de nos jours plusieurs mécanismes permettant de par exemple traiter en parallèle plusieurs instructions dont certains sont détaillés ci-dessous.

II.3.2 Types de parallélisme

Quand les premiers microprocesseurs n'avaient qu'un seul cœur de calcul, le gain en performances passait par l'augmentation de la fréquence ou par la complexification du pipeline d'instructions. Mais, comme évoquées à la section I.1, ces améliorations en-

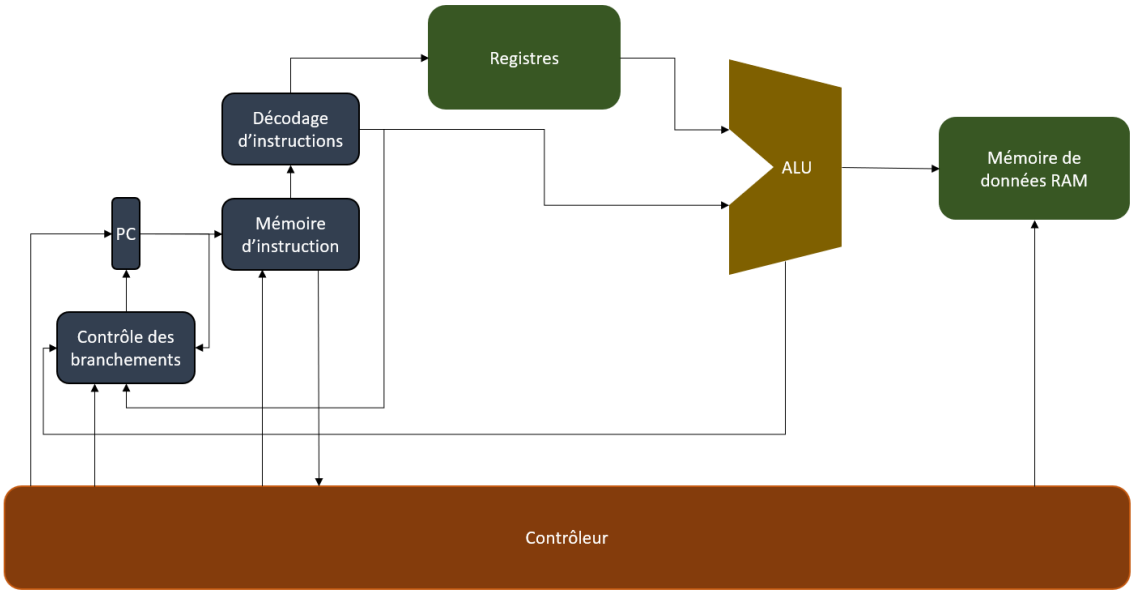


FIGURE II.2 – Architecture simplifiée d'un CPU

traînaient une forte dissipation de chaleur ainsi qu'une consommation énergétique plus importante, ce qui limite à terme cette approche. De plus, la Règle de Pollack [Danowitz et al., 2012], qui précise qu'un processeur avec G fois plus de transistors ne sera que $\sqrt{G}$ fois plus performant, réduit l'intérêt de ces approches sur les nouvelles itérations de processeurs.

Une solution pertinente consiste donc à exploiter différents types de parallélisme, et, dans le cadre de nos travaux, nous nous sommes focalisés sur ceux proposés par OpenCL, soit :

— le parallélisme d'instructions (sur un cœur) : repose sur la mise en pipeline d'instructions, et sur une prédiction efficace des branchements conditionnels,

— le parallélisme de données (sur un cœur) : consiste à effectuer une instruction sur plusieurs registres différents grâce aux unités vectorielles de certaines ALU,

— le parallélisme de threads : permet d'exécuter plusieurs piles d'instructions sur un cœur (multi-flots) ou sur différents cœurs de calculs ou processeurs.

En théorie, doubler le nombre de transistors ne permet d'avoir un gain maximal de performances que de 1.41 (Règle de Pollack), alors que répliquer un cœur permet d'augmenter les performances d'un facteur 1.8.

Tirer parti du parallélisme est donc crucial sur les CPUs modernes, et nous détaillons ci-dessous ces trois différentes approches possibles sur ces architectures.

II.3.2.1 Parallélisme d'instructions

À ne pas confondre avec le parallélisme de threads, il s'agit ici de paralléliser une suite d'instructions d'un même programme. Prenons pour illustrer ce type de parallélisme l'exemple de l'Algorithme 1 : les deux premières opérations peuvent être effectuées en

parallèle, alors que la troisième, à cause de ses dépendances, doit être exécutée une fois ces premières opérations terminées.

Algorithme 1 : Série d'instructions simples avec dépendances

```
1 c = 2 * a + b;
2 f = 3 * d * e;
3 g = f + c;
```

La difficulté principale est pour le compilateur d'identifier correctement ces types de dépendances afin d'exécuter efficacement les suites d'instructions d'un programme. Pour le développeur, il s'agit de réduire au maximum les dépendances de données, en maximisant les instructions qui peuvent potentiellement être exécutées en parallèle.

Ce type de parallélisme inclut le pipeline d'instruction, qui consiste à tenter d'avoir tous les étages internes du CPU en permanence occupés, comme illustré à la Figure II.3. Cette optimisation permet de traiter plus d'instructions en parallèle, mais ajoute toutefois une latence due à la gestion du pipeline.

Instructions \ Cycles	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
6						IF	ID

IF : Récupération d'instruction (*Instruction Fetch*)
ID : Décodage d'instruction (*Instruction Decode*)
EX : Exécution (*Execute*)
MEM : Accès mémoire (*Memory Access*)
RE : Ecriture valeur de retour dans registre (Register write back)

Tous les étages internes sont utilisés

FIGURE II.3 – Pipeline d'instructions à 5 étages.

Ce type de parallélisme repose sur la prédiction de branchement, qui spécule sur le résultat d'un branchement, et va charger les instructions correspondantes. Si la prédiction est erronée, toutes les instructions chargées par prédiction sont alors supprimées, et nous les remplaçons par les nouvelles instructions à exécuter. Les branchements conditionnels sont assez problématiques pour ce type de parallélisme, puisque la prédiction correspondante est plus complexe à mettre en œuvre.

Il faut donc veiller à réduire au minimum les branchements conditionnels ou peu prédictibles.

II.3.2.2 Parallélisme de données

Ce type de parallélisme sur CPU n'est possible que si l'ALU correspondante intègre des unités vectorielles de calcul. Dans le cas où l'ALU a bien des unités vectorielles comme les Streaming SIMD Extensions (SSE) ou les Advanced Vector Extensions (AVX), il est possible d'exécuter la même opération sur plusieurs registres en même temps. Toutefois, même si certains compilateurs prennent en charge automatiquement la vectorisation de certaines instructions, la mise en œuvre de ce type de parallélisme sur CPU est complexe et a de nombreuses limitations.

II.3.2.3 Parallélisme de threads

Exploiter ce parallélisme avec un CPU multi-cœur repose sur un principe simple : chaque cœur exécute ses propres piles d'instructions, que l'on appelle des threads. L'avantage majoritaire est que ces derniers sont gérés par le système d'exploitation, et que chacun d'entre eux communique par la mémoire partagée. De plus, la définition de threads est relativement simple du point de vue du développeur, mais sa complexité réside dans la manière d'accéder aux ressources partagées.

En effet, en choisissant de découper une multiplication matricielle entre différents threads par exemple, il faut veiller à garantir que ces derniers se partagent correctement les données afin d'éviter les conflits de lectures et d'écritures en mémoire, et gérer l'arbitrage le cas échéant.

II.3.3 Performances théoriques

Ainsi, les performances théoriques d'un CPU dépendent d'un certain nombre de caractéristiques techniques, notamment de la présence d'unités vectorielles, mais aussi de la fréquence, du nombre de cœurs, de la taille des mémoires caches, ou encore de l'efficacité du pipeline.

Si l'on considère le CPU d'Intel Xeon E5-2667, qui a 6 cœurs cadencés à 2.9 GHz, il intègre bien des unités vectorielles de type AVX (facteur SIMD de 8), et le pipeline permet en théorie d'effectuer deux additions et multiplications flottantes simple précision par cycle.

Ainsi, la performance en nombre d'opérations flottantes simples précisions est donnée par (II.1).

$$Performance_{FP32}(GFLOPs) = \#AddMult * \#Vect * \#Coeurs * Frequence(GHz) \tag{II.1}$$

où $\#AddMult =$ nombre d'additions et multiplications simultanées (pipeline),
$\#Vect =$ facteur de vectorisation,
$\#Coeurs =$ nombre de cœurs,
$Frequence =$ fréquence (GHz).

Pour le CPU considéré, nous obtenons donc une performance de $(2*2)*8*6*2.9 = 556.8$ GFLOPs

En réalité, ces performances sont peu souvent atteignables, puisqu'il faut, tout au long du programme, avoir des opérations qui permettent d'être vectorisées d'un facteur 8 en plus de pouvoir effectuer deux additions et deux multiplications par cycle.

Suivant les spécificités des implémentations, il est possible de calculer une performance réduite plus cohérente. Par exemple, si la vectorisation n'est pas utilisée, la performance maximale théorique en est alors divisée par 8.

II.3.4 Évolution future : RISC-V

Face aux multiples ralentissements des performances ressenties par l'industrie des semi-conducteurs, une évolution des architectures CPU, le RISC-V a récemment vu le jour. Initialement développé à l'Université de Berkeley en Californie en 2010, ce standard s'appuie sur le principe du RISC, et les deux aspects novateurs de cette nouvelle itération sont, d'une part, son modèle économique, et d'autre part, sa modularité [Asanović and Patterson, 2014].

Parce que l'expertise nécessaire à la conception d'une architecture de jeu d'instruction nécessite des compétences dans un très grand nombre de domaines, de l'électronique numérique à la création d'un compilateur, la grande majorité des fournisseurs d'architectures informatiques touchent des redevances sur les plans, brevets, et droits d'auteurs qui en découlent. Liés à ces architectures, il y a souvent des accords de confidentialité pour restreindre la divulgation de ces propriétés intellectuelles, de sorte qu'il est assez complexe de comprendre pourquoi certains choix d'architectures sont meilleurs que d'autres, et l'opacité peut conduire à des failles majeures de sécurité, comme nous avons pu le voir avec Spectre et Rowhammer [Mutlu, 2019], où les défauts de conceptions des CPUs affectés rendent vulnérables tout système d'exploitation utilisant ces processeurs.

Un standard ouvert permet de mitiger ce genre de failles, puisque la communauté peut plus facilement les repérer et y apporter des correctifs. C'est dans ce contexte que le RISC-V se propose comme un standard libre et ouvert de jeux d'instructions modulables, dont la réalisation physique d'un processeur supportant cette ISA n'est toutefois pas nécessairement libre ni ouverte.

Il y a actuellement quatre jeux d'instructions de bases :

- RV32E : jeu d'instruction 32 bit réduit pour le domaine de l'embarqué,
- RV32I : jeu d'instruction 32 bit classique,
- RV64I : jeu d'instruction 64 bit,
- RV128I : jeu d'instruction 128 bit,

Ceux-ci sont modulables par un très grand nombre d'extensions différentes, dont :

- F : pour le calcul flottant simple précision,
- D : pour le calcul flottant double précision,
- J : pour les langages interprétés (Java, Ruby, Scala, R, ...),
- V : pour les opérations vectorielles.

Comme les jeux d'instructions de base n'ont pas de couche d'architecture supplémentaire, comme la prise en charge de l'exécution dans l'ordre ou dans le désordre, il est possible d'implémenter facilement la prise en charge d'autres architectures, telles que les FPGAs, ou les GPUs.

Micro Semi a par exemple intégré dans un Système sur puce (*System on Chip*) (SoC) deux CPUs RISC-V avec un FPGA [Microsemi, 2018].

Cette approche modulaire connaît un certain engouement, à l'heure où le monopole d'Intel et de ARM est remis en question, et Western Digital a par exemple dévoilé sa volonté de n'intégrer à terme que des processeurs RISC-V dans ses serveurs de stockage.

De plus, les différents commerciaux entre les Etats-Units d'Amérique et la Chine ont poussé certains grands constructeurs à s'affranchir des propriétés industrielles classiques en développant des CPUs à base de RISC-V également, à l'image d'Alibaba qui a présenté en juillet 2019 le XT910 [Alibaba, 2019]. Ce CPU gravé en 12nm embarque 16 cœurs cadencés à 2.5 GHz, et supporte à la fois des instructions en 16, 32, et 64 bits.

Pour conclure, le RISC-V est une approche pertinente qui fournit, à l'heure ou l'Adéquation Algorithme Architecture est inévitable, de la modularité aux CPUs qui n'en comportaient pas suffisamment.

II.4 GPU : parallélisme massif

Dans cette section, nous allons présenter certains concepts importants des architectures GPUs, en prenant exemple sur les cartes du constructeur Nvidia.

II.4.1 Principe

Même si les architectures pour optimiser le rendu graphique sur les ordinateurs existaient déjà avant, le terme *GPU* apparaît pour la première fois pour désigner le processeur graphique de la Playstation 1, commercialisée en 1994.

Portée par le domaine du rendu graphique, l'architecture interne d'un GPU classique ressemblait à celui illustré à la Figure II.4, qui permettait, à partir d'une caractérisation d'un nuage de points, d'afficher visuellement des formes géométriques paramétrables.

En 2001, avec la famille des GeForce 3, Nvidia révolutionne l'industrie du jeu vidéo et propose une architecture avec un pipeline graphique en partie programmable et une interface de programmation d'application étoffée pour le programmer efficacement. Il faudra néanmoins attendre 2006, avec le GPU GeForce 8800 pour que Nvidia introduise un nouveau langage de programmation pour ses cartes, CUDA [Nvidia, 2006]. L'idée derrière ce langage était de permettre aux développeurs de tirer parti des puissances de calculs des GPUs pour faire autre chose que de l'affichage vidéo. Il était alors possible d'implémenter des algorithmes généraux sur GPU en utilisant ce langage, dont la syntaxe est assez proche du C.

De nos jours, les GPUs sont utilisés dans de très nombreux domaines, comme les ordinateurs bureautiques, les consoles de jeux, les systèmes embarqués, les téléphones, ou encore les calculs intensifs dans des serveurs haute performance.

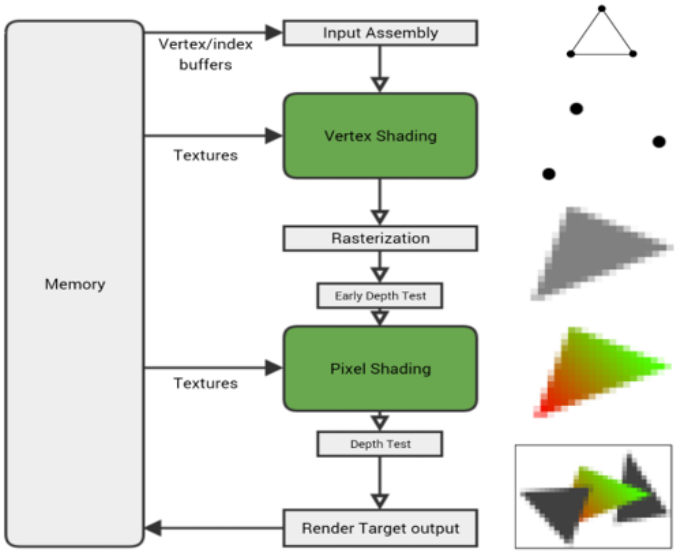


FIGURE II.4 – Pipeline graphique d'un GPU

Source: fragmentbuffer.com

II.4.2 Architecture

La Figure II.5 permet d'illustrer l'architecture d'un GPU Nvidia moderne.

Il est constitué d'un ensemble de Streaming Multiprocessors (SM), eux-même constitués de cœurs de calcul. Ainsi, un GPU est composé d'un très grand nombre de cœurs [3], et ces derniers ont un jeu d'instruction réduit par rapport aux CPUs, mais permettent néanmoins d'effectuer rapidement les opérations mathématiques usuelles.

L'une des particularités des GPUs est que pour chaque SM, il existe des blocs internes, illustrés à la Figure II.6, qui permettent de manipuler des données en parallèle des étapes de calcul.

Optimiser un algorithme sur GPU consiste donc à maximiser le débit de calcul en veillant à :

— avoir suffisamment d'instances indépendantes d'un problème pour utiliser tous les cœurs de calcul du GPU considéré et tirer parti du parallélisme massif de l'architecture,

— recouvrir le temps de transfert des données par des calculs.

II.4.3 Performances théoriques

Disposant d'un très grand nombre de cœurs, la stratégie d'optimisation d'un algorithme sur GPU passe par l'utilisation d'un nombre important d'instances différentes d'un même programme, ce qui implique que le problème à résoudre peut être subdivisé en un nombre de groupes supérieur au nombre total d'unités de calcul.

3. Plus de 5000 dans certains GPUs actuels.

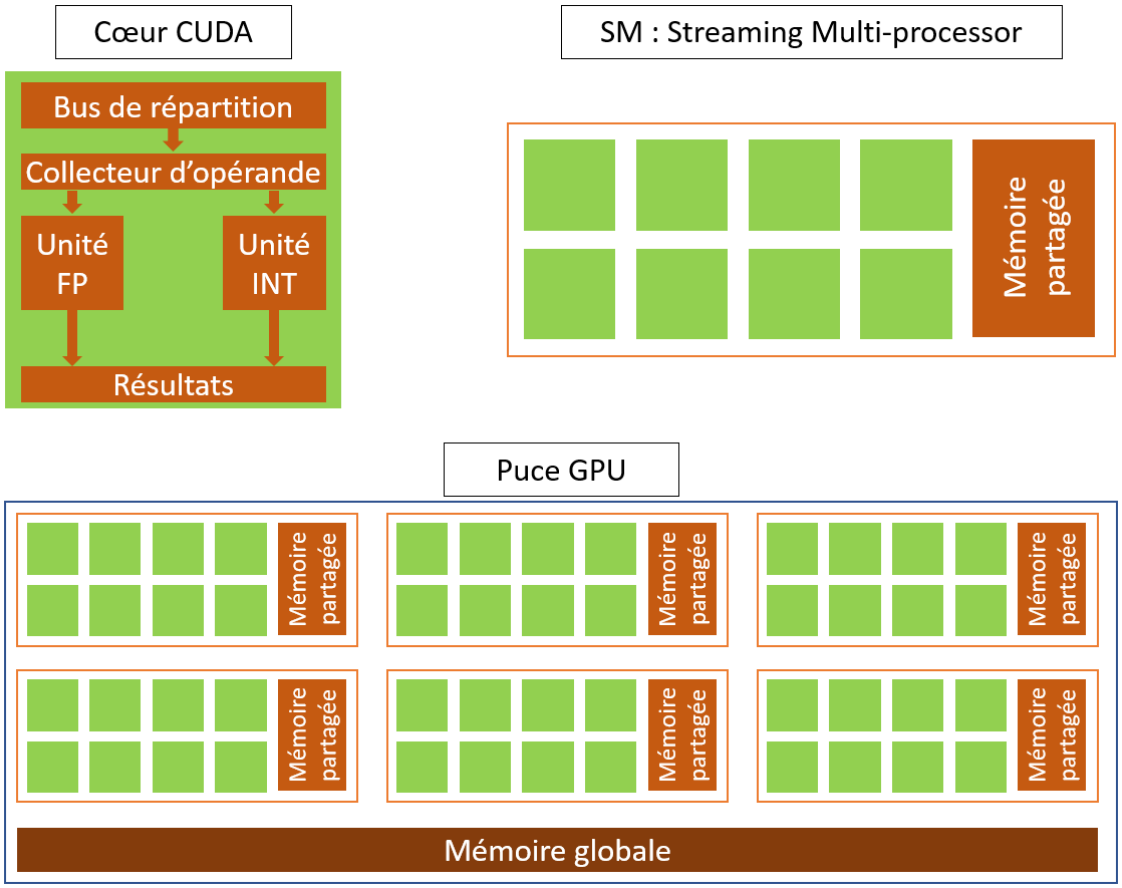


FIGURE II.5 – Architecture interne simplifiée d'un GPU Nvidia

Les performances théoriques d'un GPU sont calculées de manière assez similaire à celles d'un CPU, à cela près que le découpage des cœurs est différent entre ces deux architectures.

En s'appuyant sur les notations illustrées à la Figure II.5, la formule (II.2) nous donne donc les performances théoriques maximales d'un GPU en simple précision.

$$Performance_{FP32}(GFLOPs) = \#AddMult * \#SM * \#C_{SM} * Frequence(GHz) \quad \text{(II.2)}$$

où $\#AddMult =$ nombre d'additions et multiplications simultanées,
$\#SM =$ nombre de Streaming Multi-processor,
$\#C_{SM} =$ nombre de cœurs par SM,
$Frequence =$ fréquence (GHz).

Pour l'exemple, prenons le GPU 1080 Ti de Nvidia. Cadencé à 1.48 GHz, il a 28 SMs contenant chacun 128 cœurs CUDA. Chaque unité pouvant effectuer deux opérations flottantes par cycle, nous obtenons donc une performance simple précision de $2 * 28 *$

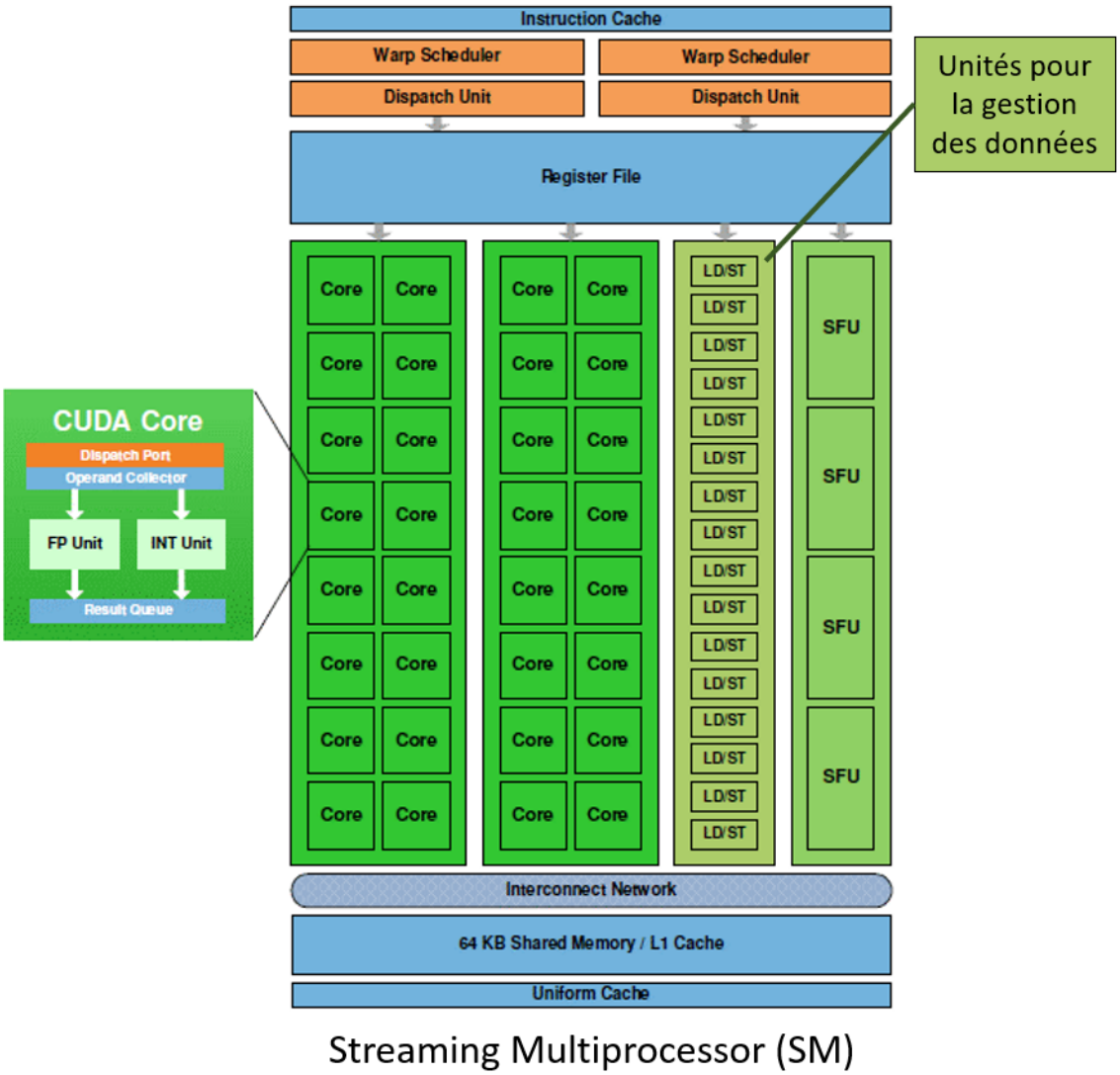


FIGURE II.6 – Détails d'un SM GPU - Cœurs de calcul et unités de gestion des données

Source: Nvidia

$128 * 1.48 = 10'609$ GFLOPs soit 10.61 TFLOPs

Ces performances théoriques ne sont atteignables que si toutes les ALUs sont utilisées à leur plein potentiel, et, encore plus sur les GPUs, il est nécessaire d'avoir une structure mémoire avec un débit suffisant pour alimenter tous les cœurs élémentaires de calcul en parallèle.

II.4.4 Langages de programmation

La programmation des GPUs peut se faire à différents niveaux d'abstraction, mais tous les langages correspondants s'appuient sur une vision SPMD (Single Program, Mul-

tiple Data), où le problème est découpé en threads élémentaires traitant de nombreuses données différentes. Ce modèle de programmation à grain fin permet d'utiliser différents types de parallélismes à plusieurs niveaux d'une architecture GPU.

Nous avons en premier lieu la **programmation de haut niveau**, qui regroupe les outils ou extensions intégrés dans des logiciels spécialisés (Tensorflow, Matlab Parallel Computing Toolbox, Indesign, ...). A un niveau assez haut mais avec une certaine gestion du matériel, nous pouvons citer les langages de **programmation par directives**, comme OpenACC [OpenACC, 0] ou encore OpenMP [OpenMP, 0]. Et finalement, il est possible de les programmer avec des **langages bas niveau**, comme CUDA ou OpenCL (voir section II.6).

II.5 FPGAs : plateforme reprogrammable

II.5.1 Historique et évolution

Les FPGAs sont des circuits intégrés conçus pour être reconfigurés à souhait après fabrication. Depuis 1969 avec le tout premier MPGA (*mask-programmed gate array*), le XC157 de Motorola, ces architectures sont très largement utilisées dans des domaines spécifiques comme les systèmes embarqués [Garcia et al., 2006] ou encore les systèmes critiques [Wegrzyn, 2001].

Ces dernières années, la demande pour des centres de données toujours plus importants, en plus d'une loi de Moore à bout de souffle, remettent sur le devant de la scène cette technologie pour des utilisations grand public. En effet, ces plateformes, en plus de pouvoir être reprogrammées, permettent une maîtrise de la consommation énergétique inégalée par rapport aux architectures conventionnelles concurrentes comme les CPUs et les GPUs. Aussi, certains analystes du domaine voient le domaine des serveurs et des centres de données comme le premier marché pour les FPGAs dans la décennie à venir [Black, 2019]. C'est l'une des raisons qui expliquent pourquoi Intel s'est porté acquéreur d'Altera, le second constructeur de FPGAs, le 28 décembre 2015, afin de consolider sa position dans ce marché à très fort potentiel.

Le principe de fonctionnement est simple. Quelle que soit la fonction informatique complexe, elle peut être traduite en une série d'opérations logiques élémentaires (telles que les AND, OR, NOT, ...). Les FPGAs intègrent un grand nombre de blocs logiques qui peuvent être reconfigurés afin d'obtenir le comportement voulu.

Ainsi, à partir d'une description fonctionnelle d'une application, les outils FPGAs génèrent un Bitstream. Ce fichier, qui est en fait une succession de bits, contient les données de configuration nécessaires pour reprogrammer les connexions et les éléments reprogrammables du FPGA afin d'implémenter la fonction désirée. Une fois mis sous tension, cette configuration est chargée et vérifiée, et le FPGA est alors considéré comme correctement configuré [4].

Ces architectures reconfigurables ont la particularité depuis quelques années d'intégrer de très nombreuses portes logiques et blocs mémoires (mémoire vive (*Random Access Memory*) (RAM)) pour implémenter des calculs numériques avancés, mais aussi des

4. Il est possible de reconfigurer un FPGA sans devoir le mettre hors tension, et cette notion s'appelle la reconfiguration dynamique.

fonctionnalités analogiques permettant d'y effectuer des opérations couramment utilisées en traitement du signal. De nombreux outils à différents niveaux d'abstraction permettent ainsi de configurer ces plateformes.

Nous retrouvons en premier lieu l'utilisation de langages de description matérielle (« Hardware Description Language ») tels que Verilog ou VHDL. Au début du développement sur FPGA, ces langages ont su devenir les langages principaux pour la conception d'algorithmes s'exécutant sur FPGA. Il s'agit ici de décrire et d'élaborer le circuit sur un modèle de flux de données, où l'on connecte une série de blocs entre eux par le biais de signaux. La vérification d'une conception à l'aide de ces langages se fait en écrivant des tests qui permettent de valider le comportement de l'architecture en vérifiant les sorties suivant les entrées paramétrées.

Le principal défi de ce mode de programmation est qu'elle requiert une expertise pointue détenue par des ingénieurs et des chercheurs au profil très spécialisé, ce qui est l'une des raisons pour laquelle la technologie FPGA était peu utilisée par la grande majorité des scientifiques et des ingénieurs.

Assez rapidement, de nouveaux outils ont vu le jour, se basant sur la synthèse de haut niveau (« High-level synthesis », HLS). Ces processus automatiques permettent d'interpréter une expression haut niveau des spécifications d'un algorithme (souvent écrite en langage C/C++, System C, ou Dataflow), en la transformant en une implémentation matérielle fidèle aux contraintes imposées.

Cette solution a connu un engouement particulier, car l'un des enjeux principaux du domaine est de pouvoir implémenter efficacement et rapidement des algorithmes sur FPGA. Il est possible de citer des outils académiques, comme AUGH [TIMA, 2012], GAUT [Coussy et al., 2008], CHiMPS [Putnam et al., 2008], mais il existe également de nombreux outils industriels comme ROCC [ROCCC, 2013], LegUp [Canis et al., 2011] Catapult [Bollaert, 2008]. Nous retrouvons logiquement les ateliers de développement de Xilinx et d'Intel : Vivado HLS et Intel HLS Compiler. Certains outils, à l'image de PREESM [Pelcat et al., 2014], acceptent également comme spécification d'entrée un graphe en flot de données.

Toutefois, pour un ingénieur logiciel, même s'il retrouve dans ces outils la syntaxe linguistique du C, l'expertise nécessaire pour optimiser un algorithme est équivalente à celle d'un ingénieur matériel, et la version optimisée d'un code ayant pour cible un FPGA à l'aide des outils HLS diverge très souvent d'un code développé pour un processeur classique.

Dernièrement, le regain d'intérêt pour les technologies FPGAs a poussé les industriels du domaine à fournir des outils visant à simplifier au maximum leur programmation, en déportant le savoir-faire matériel dans l'élaboration des ateliers de transformation automatique du code, afin de permettre une description au niveau logiciel de ces architectures.

Xilinx propose une suite logicielle générique du nom de SDx, pour « *Software Defined* - définition logicielle », qui permet à partir de programmes en C, C++, ou OpenCL, de générer un Bitstream pour ses FPGAs. Cette suite logicielle se décline en deux catégories, SDSoC qui est optimisée pour les SoC, et SDAccel, qui permet de programmer les FPGAs discrets, connectés à la carte mère via une liaison PCIe.

Intel de son côté propose un kit de développement : Intel FPGA SDK for OpenCL. Ce dernier comprend lui aussi des librairies personnalisées ainsi qu'un framework, mais n'inclut pour le moment pas d'interface de développement. La programmation se fait ex-

clusivement en C/C++ du côté CPU, et en OpenCL pour le FPGA. Il est à noter que cet outil permet aussi bien de générer un code pour un FPGA intégré dans un System On chip (SoC) comme pour ceux reliés à la carte mère via une liaison PCIe.

Nous reviendrons plus en détail sur les différents flots de conception d'un algorithme sur FPGA à la sous-section II.5.3, mais l'engouement pour ces nouvelles approches est clair, et se décline en toujours plus de contributions [Cadenelli et al., 2019, DiCecco et al., 2016, Shagrithaya et al., 2013, Czajkowski et al., 2012, Waidyasooriya et al., 2017].

II.5.2 Architecture usuelle des FPGAs

II.5.2.1 Vue d'ensemble

La Figure II.9 décrit la structure basique d'un FPGA type. Nous y retrouvons :

— des blocs mémoires (RAM, registres, ...),

— de la logique programmable qui permet d'implémenter des fonctions logiques (table de correspondance (*LUT - Lookup Table*) (LUT), DSP, ...),

— des éléments de routage (reprogrammables) pour connecter ces fonctions,

— des blocs d'Entrées/Sorties (également reprogrammables) qui permettent de communiquer avec les éléments autour de la puce.

Nous allons détailler dans les sous-sections suivantes le fonctionnement de chacun de ces blocs.

Il en existe également d'autres, comme les horloges et les PLLs, et la Figure II.8 montre l'architecture simplifiée d'un FPGA Intel Arria 10 avec tous les types de blocs qu'il contient.

II.5.2.2 Les blocs logiques reconfigurables (CLBs)

Comme illustré à la Figure II.9, les blocs logiques reconfigurables sont constitués de LUTs, de bascules (*Flip-Flops*), et de Multiplexeurs reliés entre eux.

Tables de correspondance (LUTs)

Les FPGAs reposent sur un type d'élément fondamental : ses portes logiques, qui sont regroupées en LUTs. Similaire à une table de vérité, les différents états de sorties possibles sont décrits de manière unique en fonction des valeurs d'entrées et du contenu de la table.

La Figure II.9 illustre l'implémentation de la fonction logique $f = (x_1 = x_2)$ avec un LUT à deux entrées.

À l'aide de trois multiplexeurs et d'une configuration interne précise, il est possible de vérifier que le schéma logique correspond bien au fonctionnement voulu de la fonction f.

Ainsi, la complexité des fonctions logiques pouvant être implémentées à l'aide de LUTs dépend de la taille de ces derniers. Dans les FPGAs modernes, il est courant d'avoir des LUTs à 6 entrées binaires.

Bascules (FFs)

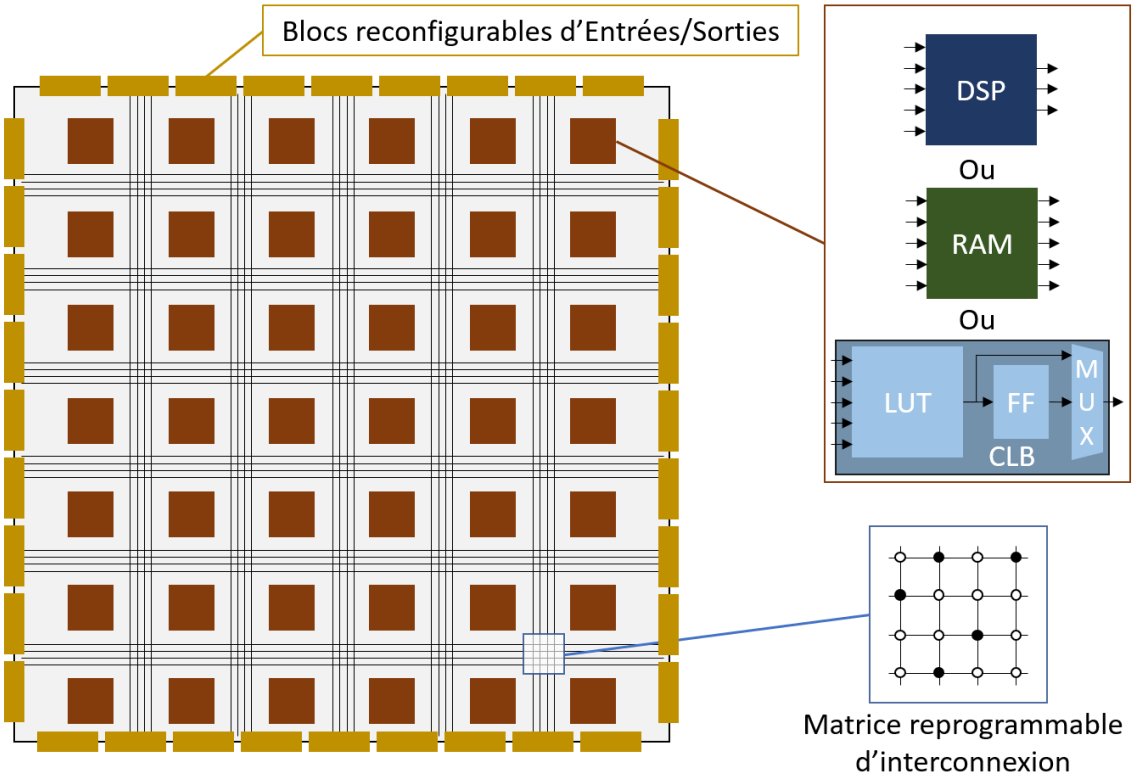


FIGURE II.7 – Structure basique d'un FPGA

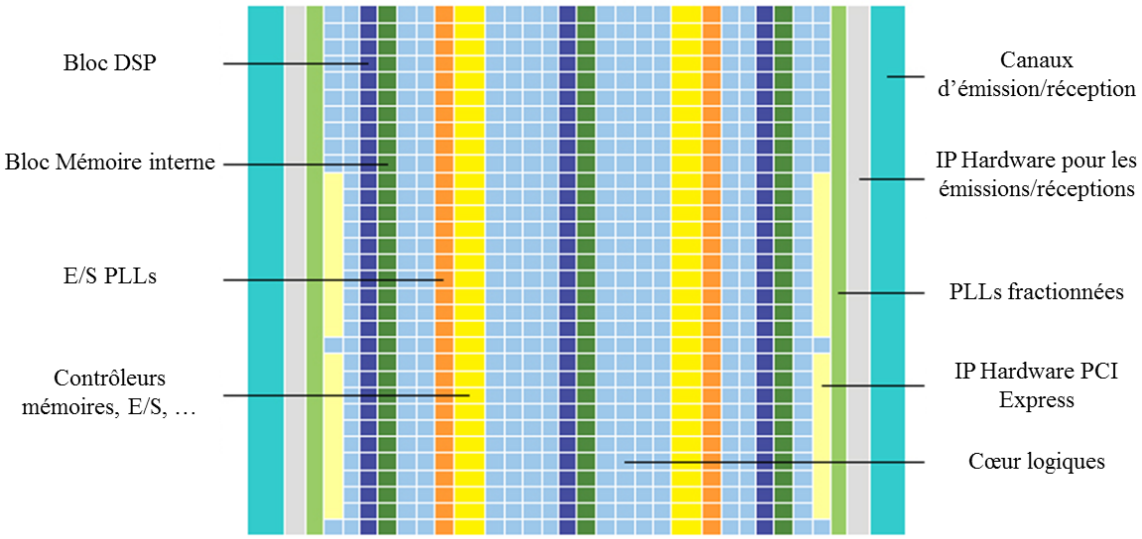


FIGURE II.8 – Architecture interne d'un FPGA Arria 10

Source: Intel

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	1

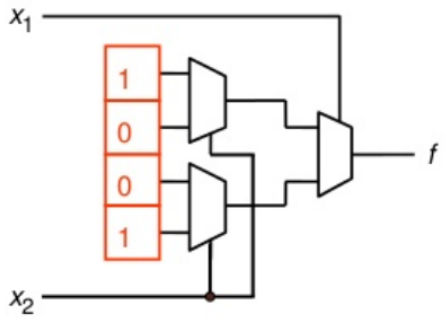


FIGURE II.9 – Implémentation de $f = (x_1 = x_2)$ en logique combinatoire (LUT à 2 entrées).

Afin de garder en mémoire les états logiques des différents calculs, ces LUTs sont couplés à des FFs. En effet, ces dernières servent à synchroniser les éléments logiques entre eux, en préservant les états logiques entre deux cycles d'horloge. Ils sont donc essentiels pour le bon déroulement des fonctions combinatoires.

II.5.2.3 Autres blocs

Mémoires RAMs

Les ressources RAMs intégrées dans la puce FPGA sont utiles pour stocker ou faire transiter les données entre différents étages de calculs. Même s'il est toujours possible de stocker des tableaux dans les bascules des LUTs, cette utilisation s'avère coûteuse en ressource, et c'est pourquoi les FPGAs modernes intègrent de plus en plus de blocks RAM pour soulager les CLBs-.

Blocs DSPs

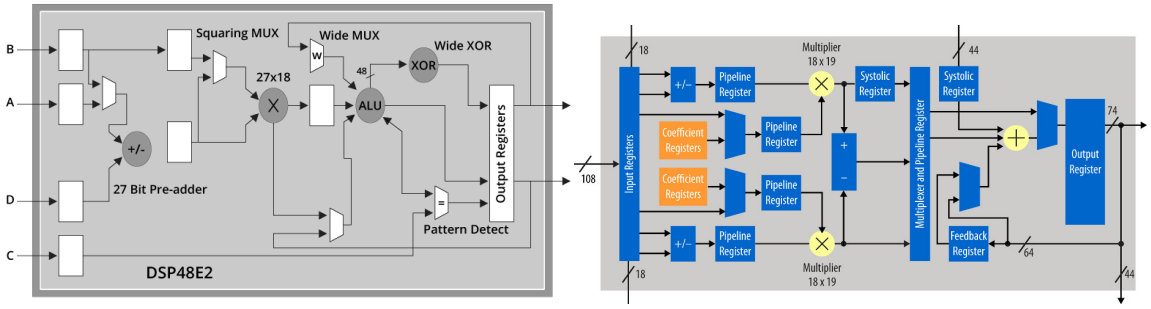
Si l'addition ou la soustraction binaire est relativement simple, la multiplication à l'aide de logique combinatoire est bien plus complexe. Ainsi, les FPGAs intègrent des DSPs similaire à ceux de la Figure II.10, pour éviter que des multiplications flottantes ne consomment trop de LUTs et de FFs.

Blocs d'Entrées/Sorties

Ces blocs, présents sur toute la périphérie du FPGA, permettent de faire le lien entre les broches du composant et le circuit interne du FPGA. Cette interface avec le reste de la plateforme peut être reconfigurée de différentes manières, soit en entrée pour déclencher un traitement suivant la réception d'un signal d'horloge par exemple, soit en sortie vers des périphériques externes.

Matrices d'interconnexions

Comme la taille des CLBs est limitée, il arrive souvent qu'une fonction logique complexe nécessite plusieurs CLBs en parallèle pour son implémentation. Ces matrices d'in-



(a) FPGAs Xilinx Ultrascale

Source: Xilinx

(b) FPGAs Intel Stratix 10

Source: Intel

FIGURE II.10 – Blocs DSP de deux FPGAs récents

terconnexions reconfigurables permettent donc de lier plusieurs entrées et sorties de différents blocs logiques pour en étendre les fonctionnalités.

II.5.3 Flots de conception FPGA

Nous présentons dans cette section les différents flots de conception d'une application sur FPGA, illustré à la Figure II.11.

Que nous choisissions d'exprimer notre algorithme en VHDL/Verilog, C/C++, ou OpenCL, les étapes de synthèse et de placement routage ont toujours lieu. Seule la manière dont est obtenue la description langage de description de matériel (*Hardware Description Language*) (HDL) de l'algorithme diffère. En effet, dans le cas où sont utilisés les flots de synthèse de haut niveau, donc C, C++, et OpenCL, la description HDL est générée automatiquement par l'outil.

L'étape de synthèse permet de découper les fonctions à implémenter suivant les caractéristiques du FPGA cible, notamment son nombre de LUTs, de FFs, de blocs RAMs, et de DSPs. À la fin de cette étape, une Netlist [5] est générée.

Le placement consiste alors à choisir, notamment à partir des contraintes des temps de propagations, la localisation idéale de chacune des fonctions de la Netlist qui permette un câblage minimal sur le FPGA. Finalement, le type d'interconnexions est choisi à l'étape de routage, suivant les contraintes de timings internes.

L'approche haut niveau de la programmation FPGA passe donc par une accélération du flot de conception, mais il est toujours possible, à partir des fichiers automatiquement générés, de les modifier pour intégrer des spécifications poussées à notre description fonctionnelle.

Toutefois, la sur-couche d'abstraction permet d'obtenir rapidement une architecture fonctionnelle sur FPGA, mais les performances obtenues seront souvent moins efficaces qu'une implémentation à partir d'une description HDL, parce que l'ensemble des leviers d'optimisation possibles est restreint. Le compromis entre temps de développement et performance brute sera donc à évaluer suivant les contraintes du programme à accélérer.

5. Description d'un circuit électronique.

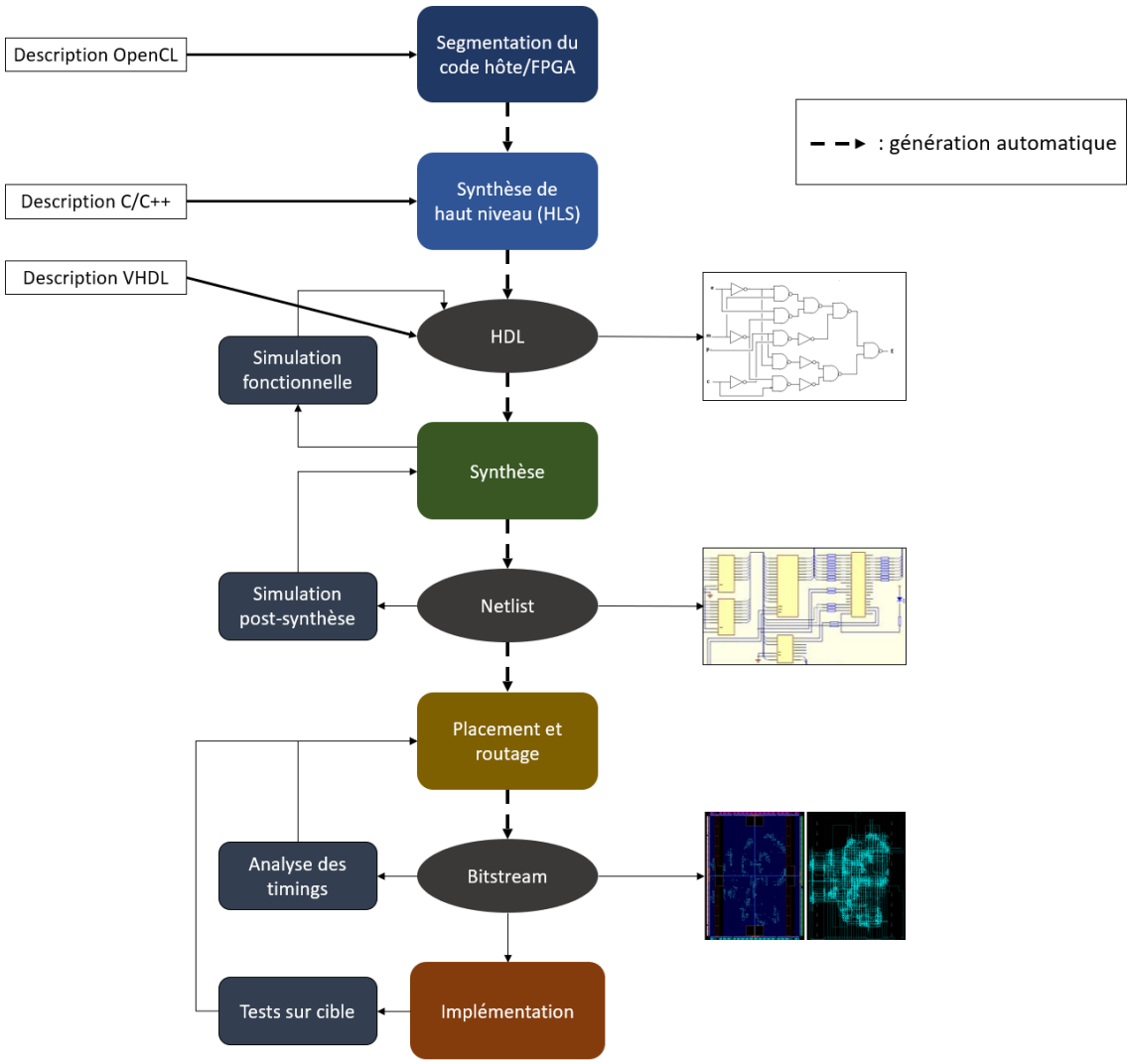


FIGURE II.11 – Flot de conception FPGA

II.6 Standard de programmation OpenCL

Un des axes majeurs de cette thèse est d'évaluer le langage OpenCL comme outil haut niveau pour la programmation des FPGAs. C'est pourquoi nous avons jugé pertinent de présenter ici certains concepts du standard OpenCL, afin d'éviter au lecteur d'avoir trop souvent recours aux documentations techniques des constructeurs. Si nécessaire, [KhronosGroup, 2019] détaille précisément le standard OpenCL.

À l'origine initié par Apple en 2008, afin de fournir une alternative libre de droits à l'utilisation des outils Nvidia pour la programmation des GPUs, le standard OpenCL reprend donc une importante part de ces concepts, avec des directives proches du langage CUDA.

Toutefois, OpenCL a été dès l'origine pensé pour prendre en charge différentes plateformes hétérogènes en plus des GPUs, et il est donc possible de cibler des architectures diverses comme des CPUs, des GPUs, des DSPs, ou encore des FPGAs à l'aide de ce standard.

Contrairement à NVIDIA qui propose à la fois les cartes et les outils pour les programmer, OpenCL est géré par un consortium, le *Khronos Group*. Il fournit un socle commun de programmation, avec des évolutions régulières du standard, et les partenaires industriels, comme Intel, Apple, ARM, Xilinx, AMD, ou d'autres, fournissent à l'utilisateur un pilote (*ICD - Installable Client Driver*), qui prend en charge certaines versions d'OpenCL.

Pour être compatible à une version du standard, chaque vendeur doit donc implémenter toutes les fonctionnalités obligatoires de cette version. Tant que cette base est respectée, chacun est libre d'y ajouter des fonctionnalités supplémentaires.

La volonté est donc de garder une homogénéité du standard, en plus d'une évolutivité dans le temps. En effet, un code compatible avec une version d'OpenCL le sera également avec les itérations suivantes, à condition que le constructeur fournisse le pilote correspondant.

Par contre, si le standard garantit l'aspect syntaxique et conceptuel du code et des fonctions, c'est aux différents constructeurs de se charger de l'implémentation de ces fonctionnalités sur leurs architectures respectives, et les performances d'une fonction standard OpenCL peuvent grandement varier d'une plateforme à l'autre.

La principale difficulté est donc de savoir comment tirer parti au mieux d'une architecture donnée, sachant qu'OpenCL permet des styles de programmation très différents selon que l'on veuille exprimer un parallélisme de tâches ou de données par exemple.

Atelier de développement puissant, sa principale force est de pouvoir être implémenté sur presque toutes les plateformes, mais c'est aussi son plus grand inconvénient, en cela que les choix d'optimisations dépendent des caractéristiques inhérentes de l'application à implémenter et de l'architecture cible.

Ainsi, la plupart du temps, il est possible d'obtenir une portabilité fonctionnelle du code, mais pas une portabilité au niveau des performances.

II.6.1 Architecture générale

Un modèle d'architecture OpenCL, illustré à la Figure II.12, est constitué d'une plateforme hôte et de plusieurs architectures d'accélérations, appelées *devices*.

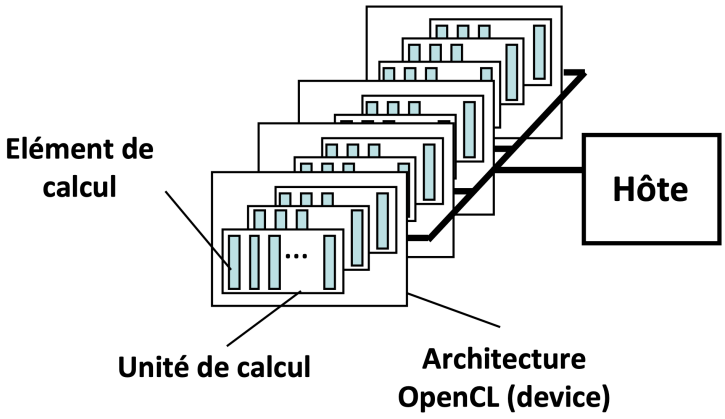


FIGURE II.12 – Modèle d'architecture OpenCL

Source: KhronosGroup

L'hôte est en charge de garantir le bon déroulage d'une exécution OpenCL, notamment en :

— vérifiant la présence des drivers adéquats pour la plateforme d'accélération,

— gérant les transferts de données entre les architectures,

— supervisant l'exécution des fonctions OpenCL sur les *devices*,

— renvoyant des codes d'erreurs en cas de problème.

II.6.2 Types de kernels

Ce standard permet de définir un découpage des problèmes, ce qui peut s'avérer utile pour des algorithmes à plusieurs dimensions[6]. Il est donc possible de définir un découpage de l'espace suivant ces dimensions, et la Figure II.13 illustre un découpage en 2D d'un problème donné.

En reprenant la Figure II.13, la grille principale, de taille $G_x * G_y$, est elle-même subdivisée sur ses deux dimensions en groupes, appelés *work-groups*, qui sont également constitués de *work-items*. Ces derniers représentent la plus petite subdivision possible du formalisme OpenCL. En découpant un problème donné en *work-items*, exécuter un algorithme correspond à exécuter toutes les instances de chacun de ses work-items.

Il est alors possible, suivant la dimensionnalité d'un algorithme donné, de choisir deux types de *kernels* OpenCL :

— Single Work-Item Kernels (SWIKs) : ce type de kernel n'a qu'un seul *work-group* ne contenant qu'un seul *work-item*. Ce type de kernel n'exprime donc pas de parallélisme de thread, et un programme ne s'exprimera qu'en une seule instance.

— NDRange Kernels (NDRKs) : ce type de programme tire parti du découpage précédent, et il y a donc obligatoirement plusieurs *work-items* pour un programme donné.

6. OpenCL ne prend en charge qu'un maximum de trois dimensions

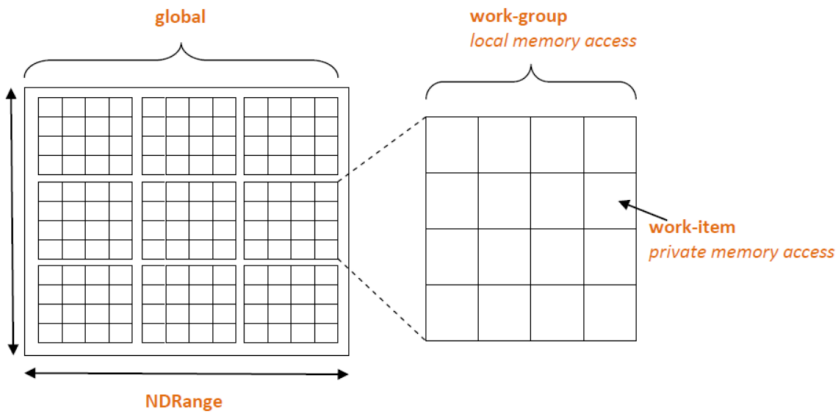


FIGURE II.13 – Découpage de l'espace sur deux dimensions

Source: KhronosGroup

D'un fonctionnement similaire au parallélisme Simple instruction multiples threads (*Single Instruction multiple threads*) (SIMT), il y a aura donc plusieurs instances de la même fonction avec des indices différents exécutées en parallèle dans un même programme.

Pour les NDRKs, la gestion des différents *work-items* par les outils OpenCL dépend du nombre de cœurs de calcul présents sur l'architecture considérée. En s'appuyant sur la Figure II.14, où nous avons supposé qu'il existait trois cœurs de calcul fonctionnels, chaque work-group va être mis dans une file d'attente, puis chargés sur un cœur dès que celui-ci est libre.

Quand un work-group est lancé sur un cœur, le standard OpenCL permet à tous les *work-items* correspondants de partager une même zone mémoire locale rapide, ce qui peut-être utile pour échanger des données entre itérations élémentaires d'un programme exploitant le parallélisme de données. Il faut cependant veiller à ne pas oublier d'implémenter des barrières de synchronisation pour éviter des conflits d'accès mémoire.

Pour les SWIKs, comme il n'y a qu'un seul work-item, il n'y a pas de gestion du découpage du problème par les outils OpenCL, et le parallélisme du programme devra s'exprimer dans le corps du kernel.

II.6.3 Architecture mémoire

Nous détaillons à la Figure II.15 le modèle d'architecture mémoire OpenCL.

En plus de retrouver le découpage en *work-groups* et *work-items* présentés à la sous-section précédente, ce modèle de hiérarchie a donc quatre niveaux de mémoires :

— *Globale* : la plus importante en termes de capacité, ce type de mémoire sert très souvent d'interface avec l'hôte. Pour en optimiser l'accès, il faut favoriser l'accès aux données par paquets afin de maximiser le débit des bus de données.

— *Constante* : Mémoire optimisée pour les succès de cache. De taille variable suivant les architectures, il n'est possible d'y stocker que des données en lecture seule.

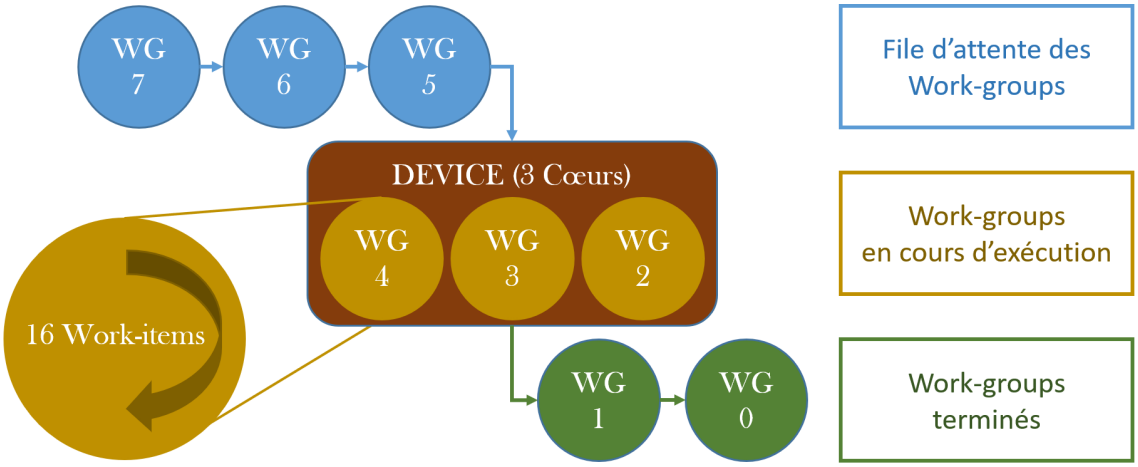


FIGURE II.14 – Exécution d'un programme avec 16 *work-items* par *work-group* sur trois cœurs de calcul

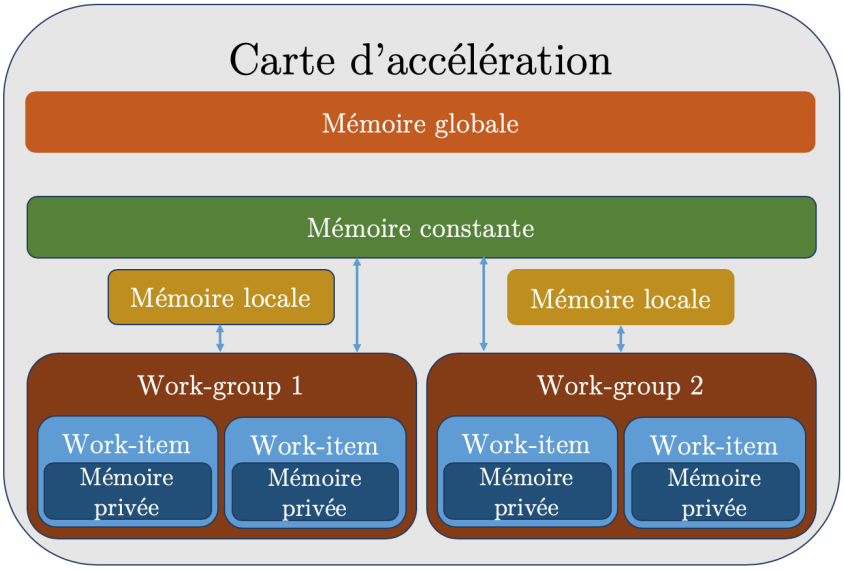


FIGURE II.15 – Architecture mémoire OpenCL

— *Locale* : Mémoire de quelques octets, les données sont partagées entre tous les *work-items* d'un même *work-group*. Le temps d'accès est identique quelque soit la localisation de la donnée dans cette mémoire.

— *Privée* : Mémoire de quelques bits, les données ne sont pas partagées et ne sont donc accessibles que par le *work-item* correspondant.

II.7 Conclusion

Dans cette partie, nous avons pu, en plus de dresser un constat de l'état actuel de l'industrie des semi-conducteurs, présenter les caractéristiques importantes des architectures hétérogènes CPU/GPU/FPGA, d'un point de vue :

— structurel (pipeline, jeu d'instructions, gestion de la mémoire),

— fonctionnel (type de parallélisme accès à la mémoire),

— de l'exécution, en rappelant le calcul de leurs performances.

Nos travaux se sont focalisés sur l'évaluation de l'approche basée sur le langage OpenCL sur les cibles FPGAs, et notre contribution principale consiste à proposer une méthodologie d'accélération d'algorithmes en OpenCL sur FPGAs vus comme des accélérateurs matériels.

Deuxième partie

Méthodologie AAA pour le co-processing sur FPGA en OpenCL

Remarques introductives

Nous proposons tout d'abord au Chapitre III notre modélisation d'un FPGA en co-processing, en définissant des métriques de performance sur FPGA. En particulier, nous adaptons un modèle asymptotique de performances, le *Roofline Model*, en lui adjoignant un paramètre qui reflète la réplication effective d'un algorithme sur une carte FPGA en fonction de ses ressources logiques et de sa bande passante.

Le chapitre IV est consacré à la présentation et à la caractérisation de certaines optimisations OpenCL, que nous mettons en forme dans une démarche structurée au Chapitre V. Dans ce dernier, nous détaillons notre stratégie d'optimisation, en introduisant tout d'abord la notion d'optimum de *Pareto* pour l'optimisation des performances sous contraintes, pour finalement mettre en forme notre méthodologie d'accélération d'algorithmes en OpenCL sur FPGA.

Chapitre III

Modélisation d'un FPGA : Roofline et métriques

Sommaire

Nous avons décidé de choisir comme architecture principale la technologie FPGA pour l'ébauche d'une méthodologie d'adéquation algorithme architecture, et notamment son utilisation en tant que plateforme d'accélération en tant que co-processeur.

Néanmoins, nous nous proposons ici de résumer les principaux attraits des architectures les plus communes, tout en explicitant les grandes lignes qui pourraient justifier le choix de l'une d'entre elles.

De façon préliminaire, nous mettons en lumière les principaux attraits des architectures les plus courantes, pour enfin détailler notre modélisation ainsi que notre démarche d'optimisation de programmes OpenCL sur FPGA.

III.1 Choix de l'architecture

Si nos travaux se sont focalisés sur les nouveaux outils de configuration des FPGAs, ces plateformes, comme toutes les autres, ne sont efficaces que pour une certaine catégorie d'algorithmes dans un contexte HPC.

Aussi, nous commençons par résumer les principaux attraits des architectures les plus communes, qui ont été présentées dans le chapitre II et discutons de l'opportunité de les choisir, dans le cadre de notre travail.

Cette discussion, peut permettre d'accompagner en amont la réflexion générale sur la définition de l'architecture finale d'un projet.

CPU :

L'avantage principal du CPU, et ce qui explique sa très grande diffusion, est sa polyvalence, non seulement en ce qui concerne les instructions élémentaires réalisables mais aussi en ce qu'il est programmable par de nombreux types de langages.

Un second avantage est la possibilité d'exprimer un parallélisme de tâche et dans une moindre mesure un parallélisme de données, notamment sur les dernières générations de CPU qui intègrent des cœurs multiples.

Un troisième avantage est la fréquence élevée de ces architectures, canalisant les efforts dans ce sens du domaine.

L'inconvénient principal de cette architecture est que, étant une machine de Turing [Turing1936], l'exécution à la suite des différentes instructions pénalise les performances des applications, mais cela est en partie contrebalancé par l'utilisation de fréquences toujours plus hautes.

En conclusion, le CPU est la plateforme de référence quand il s'agit d'obtenir rapidement et relativement simplement une première implémentation d'un algorithme, quelque soit le niveau de représentativité voulu.

GPU :

Contrairement au CPU, le GPU a un jeu d'instruction assembleur réduit, ce qui signifie que les unités de traitement en sont plus simples et donc plus rapides.

Le principal avantage du GPU, réside dans le nombre important de cœurs de calculs flottant, de l'ordre de grandeur du millier, avec certaines cartes prenant en charge la double précision.

Son potentiel pour paralléliser massivement des calculs est donc bien réel, si tant est que l'on garde à l'esprit les restrictions qu'imposent par exemple, la loi d'Amdhal.

Ses trois inconvénients principaux sont :

— le jeu d'instruction réduit qui permet d'interpréter moins de types d'opérations,

— une architecture matérielle imposant un parallélisme de threads dans la majorité des cas,

— une fréquence de deux à trois fois plus faible que sur les CPUs équivalents.

La vaste adoption de ces architectures a notamment permis le développement de librairies optimisées, spécialisées dans le traitement d'images, dans les calculs scientifiques, et plus récemment dans le domaine de l'intelligence artificielle. Elle dispose de plus, de méthodes de programmation comme OpenMP ou OpenACC [1], qui permettent de gagner en abstraction à l'aide de directives préprocesseur, et évitent de devoir rentrer dans les détails de la programmation matérielle avec CUDA ou OpenCL.

NPU :

Cette architecture récente a été conçue pour l'apprentissage automatique et pour les réseaux de neurones, spécialisée dans les calculs matriciels à très fort volume, elle ne contient pas de pipeline graphique comme il en existe dans les GPUs.

Ainsi, son avantage majeur est de pouvoir répondre aux besoins de calculs des mécanismes d'apprentissage et d'inférence des réseaux de neurones de manière efficace.

De son adéquation à ce domaine découle son principal inconvénient : son absence de flexibilité, et ses faibles performances pour d'autres utilisations.

FPGA :

Capable de générer sur mesure une architecture pour chaque application, cette puce configurable, principalement utilisée dans les systèmes embarqués ou critiques jusqu'encore récemment, a connu un réel engouement du fait de son efficacité énergétique particulièrement adaptée à une utilisation dans les serveurs. Sa principale limitation reste la difficulté à s'approprier la connaissance du matériel, et des outils permettant d'y implémenter une application. Mais, de nombreux outils pour la configurer voient le jour, rajoutant toujours plus d'abstraction, ce qui pourrait démocratiser son utilisation.

Cette architecture reste très efficace pour des applications nécessitant une latence faible, et c'est pourquoi l'un de ses domaines majeurs d'utilisation est le domaine des communications.

L'utilisation de FPGA avec les outils haut niveau est encore limitée par le manque de librairies disponibles, mais l'intérêt pour cette technologie a entraîné de nombreuses contributions qui pourraient à terme palier à ce problème.

Autres :

Il existe de très nombreuses autres architectures spécifiques, que nous n'évoquerons pas ici. Nous pouvons néanmoins citer par exemple des puces hybrides, comme les architectures Zynq de Xilinx, qui intègrent des processeurs ARM ainsi que de la logique programmable dans un même système sur puce.

Pour résumer, chacune architecture spécifique est conçue pour répondre à un besoin précis, et la conception d'une architecture spécifique, dans le cas où les existantes ne sont pas suffisamment performantes peut avoir deux issues :

1. Ces langages permettent également de programmer des CPUs

— si le marché est important, cette architecture peut être standardisée, comme l'ont été par le passé les GPUs ou les TPUs,

— si le marché est très faible, cette architecture spécifique permettra de répondre à un cas d'étude précis et n'a donc pas vocation à être standardisée.

III.2 Modélisation d'un FPGA en *co-processing*

La reprogrammabilité d'un FPGA est l'un de ses principaux atouts, mais cette caractéristique rend ardue la définition d'une métrique de performance fiable.

A partir des propriétés inhérentes à cette architecture et afin de mieux comprendre les performances d'une implémentation sur FPGA, nous allons présenter notre modélisation d'un FPGA en coprocessing, puis nous présenterons les nouveaux paramètres que nous avons définis afin d'en améliorer les performances.

III.2.1 Système global CPU hôte + FPGA

Afin de modéliser le fonctionnement d'un FPGA dans la dynamique de co-processing permise par l'utilisation du standard OpenCL, nous allons nous appuyer sur la Figure III.1 qui dresse dans les grandes lignes les étapes qui permettent d'exécuter un algorithme sur ce type d'architecture.

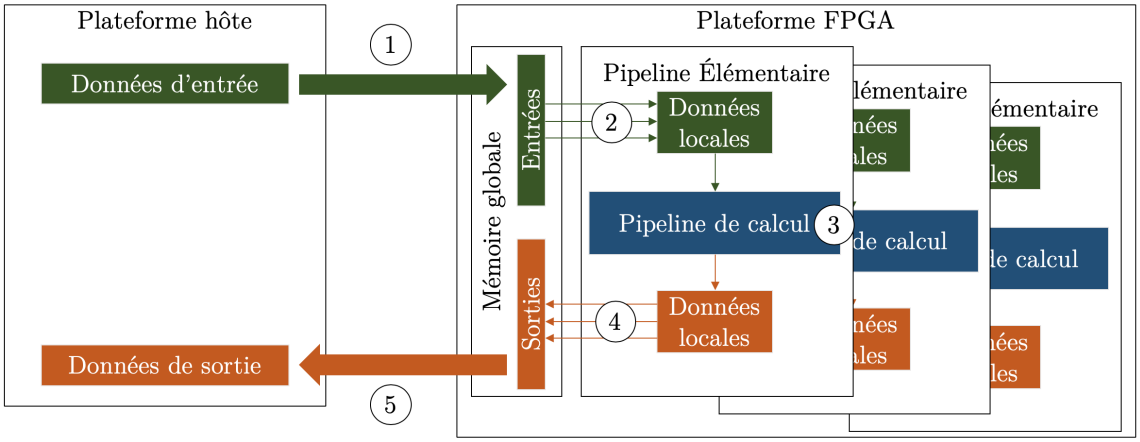


FIGURE III.1 – Étapes d'exécution des calculs sur FPGA

Dans notre modèle, la plateforme hôte est un CPU, et notre plateforme complémentaire est un FPGA.

Dans une configuration classique, les données sont sur l'hôte, ou en tout cas partagées avec celui-ci. A un moment donné, une fonction doit être exécutée sur le FPGA, et c'est alors qu'ont lieu les tâches suivantes :

Étape 1 : Copie des données d'entrée de l'hôte dans la mémoire globale du FPGA.

Étape 2 : Certaines données peuvent alors éventuellement être mises en cache.

Étape 3 : Le pipeline de calcul effectue alors les traitements demandés à l'intérieur du pipeline élémentaire.

Étape 4 : La boucle de traitement se clôt par la recopie inverse des données de sortie, de la mémoire locale à la mémoire globale FPGA.

Étape 5 : Recopie dans la mémoire de l'hôte

Ainsi, pour optimiser l'implémentation d'un programme sur une plateforme de co-processing CPU-FPGA, nous avons analysé et optimisé un certain nombre de points, tels que :

— la copie des données entre l'hôte et le FPGA

— la stratégie de stockage mémoire (types, banques, dimensionnement et partition ...)

— la mise en cache de données à l'intérieur de la fabrique FPGA et la gestion de leur politique d'accès

— la réplication des Pipelines Élémentaires, les notions de pipeline, de déroulage de boucle, et de vectorisation

— l'efficacité du pipeline de calcul

La Figure III.1 étant résolument simplifiée afin de ne pas perdre en lisibilité, il existe d'autres mécanismes, comme la communication entre fonctions implémentées sur FPGA, ou la gestion de différents types de mémoire globale, que nous aborderons plus loin dans cette section.

Nous nous focalisons donc ici sur la modélisation simplifiée d'un algorithme implémenté sur FPGA avec les outils OpenCL, et cette partie rappelle certaines notions communément utilisées dans le domaine auxquelles nous avons ajouté nos métriques personnalisées qui permettent de rapidement caractériser la pertinence des optimisations suivant les particularités de l'application et de la plateforme étudiées, afin de mesurer l'efficacité d'une implémentation.

En premier lieu, nous nous intéressons au pipeline de calcul, puis à l'éventuelle mise en cache des données dans le pipeline élémentaire. Puis, nous discutons de la réplication possible de ce dernier, afin d'aborder ensuite l'utilisation des structures mémoires de grande dimension sur les FPGAs (mémoires $global$ et $constant$), et nous intéressons, en dernier lieu, à la copie CPU $\leftrightarrow$ FPGA.

III.2.2 Pipeline de calcul

La notion de pipeline n'est pas propre aux FPGAs, mais il est nécessaire de maîtriser ce concept à défaut de ne pas pouvoir obtenir de performances suffisantes. Dans cette partie, nous allons nous intéresser à l'optimisation du cœur de calcul d'un FPGA.

Sa configuration, comme expliqué à la section II.5, consiste à traduire un algorithme en une série de fonctions logiques, qui seront ensuite implémentées à l'aide des blocs de base. La transformation d'un code ou d'une table de vérité en une architecture logique est illustrée à la Figure III.2.

Dans une architecture FPGA, le cœur élémentaire de calcul diffère d'un algorithme à l'autre, alors qu'il est statique dans l'architecture d'un CPU ou d'un GPU.

Ainsi, au lieu de devoir convertir le code source (voir section II.5) en instructions qui dépendent du jeu d'instructions du processeur cible, les outils haut-niveau sur FPGA vont créer pour chaque algorithme une architecture unique, puis effectuer un placement routage afin d'optimiser les latences et la localité des éléments à implémenter.

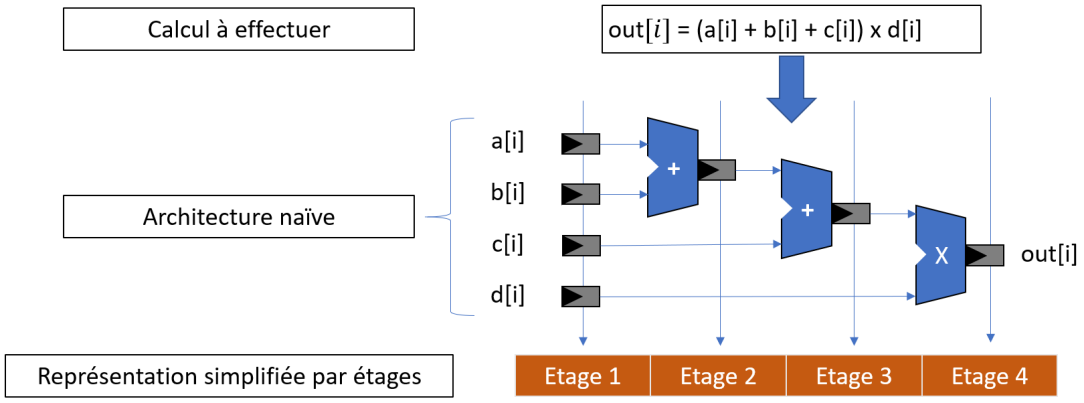
Cette dernière étape étant prise en charge par les outils des différents constructeurs, la difficulté du programmeur est d'obtenir l'architecture de calcul la plus efficace avec un nombre d'optimisations possibles limitées.

Dans ce qui suit, nous prenons l'exemple d'une architecture générée à partir d'une ligne de code avec les outils de haut niveau. Nous montrons ensuite en quoi la première architecture générée est inefficace, et proposons une manière de l'optimiser tout en définissant les métriques correspondantes.

III.2.2.1 Paramètre de performance

Considérons l'exemple simple d'une instruction de calcul additionnant en boucle les éléments de quatre tableaux, respectivement notées $a[i]$,$b[i]$,$c[i]$,$d[i]$, le calcul à effectuer étant $out[i] = (a[i] + b[i] + c[i]) * d[i]$.

La Figure III.2 illustre la génération possible d'une architecture à partir d'une ligne de code, que nous allons examiner en détail, pour expliquer en quoi cette génération est inefficace et comment il convient d'y remédier en l'optimisant.



	Cycles	Etage 1	Etage 2	Etage 3	Etage 4
	1	a,b,c,d[0]	-	-	-
	2	-	a[0]+b[0]	-	-
II = 4	3	-	-	a[0]+b[0]+c[0]	-
	4	-	-	-	(a[0]+b[0]+c[0])*d[0]
	5	a,b,c,d[1]	-	-	-
	6	-	a[1]+b[1]	-	-
II = 4	7	-	-	a[1]+b[1]+c[1]	-
	8	-	-	-	(a[1]+b[1]+c[1])*d[1]
	9	a,b,c,d[2]	-	-	-

FIGURE III.2 – Décomposition de la transformation d'un code en pipeline sur FPGA (version naïve)

Sur un FPGA, l'exécution est cadencée par le pas d'horloge, et, parce qu'il n'y a au-

cune propagation des données *c[i]* et *d[i]* tout au long du pipeline, ces dernières ne sont utilisées que tardivement dans l'exécution de cet exemple. Ainsi, au premier cycle d'horloge, les quatre données *a,b,c,d[i]* sont chargées dans des registres. Au deuxième cycle, seul le premier additionneur peut effectuer des calculs donc les valeurs de *c[i]* et *d[i]* sont toujours inutilisées, empêchant de rafraîchir les registres. Au troisième cycle, le constat est similaire. La donnée *c[i]* est enfin utilisée, mais *d[i]* bloque toujours le rafraîchissement des registres d'entrée. Ce n'est qu'au bout du quatrième cycle d'horloge que toutes les données auront été manipulées, et que de nouvelles données pourront être chargées en entrée .

Dans la Figure III.2, le tableau permet d'illustrer quels étages sont activés pendant les 9 premiers cycles d'horloge, et quels sont ceux qui sont en veille. Nous constatons bien l'inefficacité de l'architecture, puisque, sur ces 9 cycles d'horloge, et avec 4 étages de pipeline, seulement 9 cases sur 36 font des calculs. L'efficacité finale du pipeline est donc de $\frac{9}{36}$ soit de 25%.

Ce qui nous permet de définir un premier paramètre, qui est l'un des plus importants pour l'optimisation d'une exécution sur FPGAs.

Intervalle d'Initialisation (noté *II*) : c'est le nombre de cycles d'horloge nécessaire à l'alimentation incrémentale des données d'entrées.

Sur l'architecture présentée à la Figure III.2, cet intervalle d'initialisation vaut 4 (noté *II = 4*).

L'analyse du pipeline précédent montre clairement que l'inefficacité découle de l'absence de propagation des valeurs tout au long des étages de ce dernier.

Technique d'optimisation :

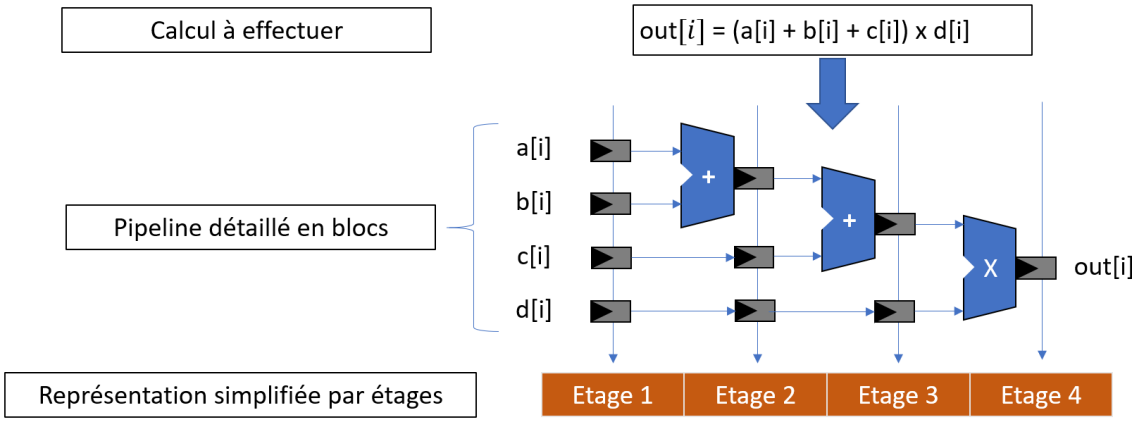
Afin de permettre d'alimenter plus efficacement le circuit, une solution efficace, que les outils de synthèse essayent d'implémenter automatiquement, consiste à propager les valeurs qui ne sont utilisées qu'en fin de calcul. Ici, il s'agit des données *c[i]* et *d[i]*. Il suffit donc, dans ce cas, d'insérer des registres pour sauvegarder ces valeurs aussi longtemps que nécessaire.

Ce mécanisme est illustré à la Figure III.3, et nous pouvons constater que l'ajout de trois registres (un pour *c*, les deux autres pour *d*) permet d'alimenter le pipeline à chaque cycle d'horloge. C'est la solution la plus optimale en termes de performance, car, une fois passé le temps d'établissement du pipeline, tous les étages effectuent des calculs à chaque cycle d'horloge.

La Figure III.3 illustre ce pipeline détaillé, et, en s'intéressant au tableau en bas de figure, l'analyse visuelle nous confirme l'efficacité théorique de notre modification d'architecture. Avec de nouveau 9 itérations à traiter par exemple, nous avons $9 * 4 = 36$ cases de calcul, 30 d'entre-elles sont utilisées, soit une efficacité de 83.3%. Nous obtenons également un Intervalle d'Initialisation de 1, ce qui est la valeur optimale.

III.2.2.2 Nombre de cycles d'un pipeline de calcul

Nous présentons ici la généralisation de l'optimisation du pipeline de calcul, en établissant la formule du nombre de cycles de calcul nécessaire au traitement de toutes les



II = 1

Cycles	Etage 1	Etage 2	Etage 3	Etage 4
1	a,b,c,d[0]	-	-	-
2	a,b,c,d[1]	a[0]+b[0]	-	-
3	a,b,c,d[2]	a[1]+b[1]	a[0]+b[0]+c[0]	-
4	a,b,c,d[3]	a[2]+b[2]	a[1]+b[1]+c[1]	(a[0]+b[0]+c[0])*d[0]
5	a,b,c,d[4]	a[3]+b[3]	a[2]+b[2]+c[2]	(a[1]+b[1]+c[1])*d[1]
6	a,b,c,d[5]	a[4]+b[4]	a[3]+b[3]+c[3]	(a[2]+b[2]+c[2])*d[2]
7	a,b,c,d[6]	a[5]+b[5]	a[4]+b[4]+c[4]	(a[3]+b[3]+c[3])*d[3]
8	a,b,c,d[7]	a[6]+b[6]	a[5]+b[5]+c[5]	(a[4]+b[4]+c[4])*d[4]
9	a,b,c,d[8]	a[7]+b[7]	a[6]+b[6]+c[6]	(a[5]+b[5]+c[5])*d[5]

FIGURE III.3 – Décomposition de la transformation d'un code en pipeline sur FPGA (version efficace)

itérations, qui permet le calcul du temps d'exécution du pipeline.

Les calculs, comme illustrés aux Figures III.2 et III.3, sont transformés en un pipeline matériel. A partir des caractéristiques de ce dernier, il est possible de prévoir la performance et le temps d'exécution approximatif de notre algorithme à l'aide des différents paramètres que nous allons détailler ci-après.

Dans un souci de lisibilité et de clarté, nous remplaçons les pipelines des figures précédentes par le pipeline de la Figure III.4 les notations des variables ainsi que leurs définitions seront introduites dans le Tableau III.1.

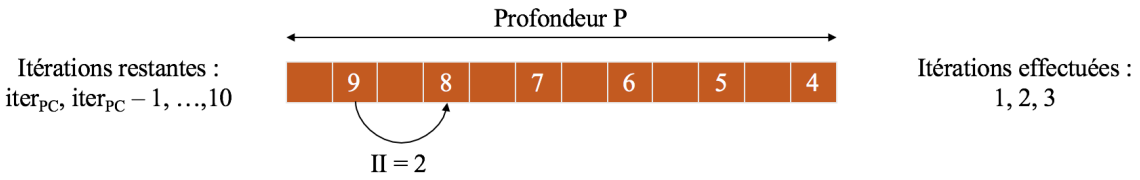


FIGURE III.4 – Vue simplifié d'un pipeline de profondeur P_{PC}, devant traiter $iter_{PC}$ itérations, avec II = 2.

Ainsi, si l'on essaye de quantifier le temps de traitement d'un pipeline, prenons le cas où nous avons $iter_{PC}$ itérations à effectuer, pour un pipeline de profondeur P_{PC}, avec un

TABLE III.1 – Pipeline : quelques définitions.

Notion	Symbole	Définition
Pipeline de calcul	*PC*	Désigne le pipeline résultant de la transformation des opérations de calcul en pipeline sur FPGA.
Profondeur	P_{PC}	Nombre de cycles dont a besoin une donnée pour traverser tout le pipeline de calcul *i.e.* le nombre d'étages de celui-ci.
Intervalle d'initialisation	II	Nombre de cycles d'horloge entre deux alimentations du pipeline.
Itérations à réaliser	$iter_{PC}$	Nombre d'itérations que doit effectuer un pipeline (dépend de l'algorithme).
Fréquence du *PC*	f_{PC}	Fréquence du pipeline de calcul.

intervalle d'initialisation de II.

La première étape est l'établissement du pipeline, c'est à dire le temps qu'il faut pour que le dernier étage du pipeline reçoive sa première donnée. Par définition de la profondeur d'un pipeline, il faut donc P_{PC} cycles d'horloge pour atteindre l'établissement du pipeline.

Une fois cette phase d'initialisation terminée, une itération sera traitée tous les II cycles d'horloge. Ainsi, il faudra au total $II * (iter_{PC} - 1)$ cycles pour que le pipeline, une fois établi, puisse traiter toutes les itérations demandées. Au final, nous établissons la formule (III.1).

Nombre de cycles d'un pipeline de calcul (noté $cycles_{PC}$) : c'est le nombre de cycles nécessaires au traitement de toutes les itérations par un pipeline de calcul.

$$cycles_{PC} = P_{PC} + II * (iter_{PC} - 1) \tag{III.1}$$

Nous pouvons maintenant nous intéresser à l'étape suivante qui consiste à déterminer le temps de traitement d'un pipeline.

III.2.2.3 Temps d'exécution du pipeline de calcul

Si l'on ne considère que le pipeline de calcul, nous pouvons modéliser le temps d'exécution[2] de ce dernier, par le rapport du nombre de cycles nécessaires au traitement de toutes les itérations et de sa fréquence (III.10)

$$T_{PC} = \frac{cycles_{PC}}{f_{PC}} = \frac{P_{PC} + II * (iter_{PC} - 1)}{f_{PC}} \tag{III.2}$$

La section suivante reprend et généralise ces différents concepts, en prenant en compte la spécificité des mémoires locales.

2. en reprenant les notations présenté au Tableau III.1

III.2.3 Pipeline élémentaire

III.2.3.1 Composition

Le pipeline élémentaire englobe le pipeline de calcul, ainsi que les variables locales et privées de l'algorithme correspondant, car ses structures mémoires sont liées à un pipeline de calcul unique. En effet, dans le cas d'une réplication du pipeline de calcul, les objets mémoires [3] correspondants sont également répliqués.

C'est pourquoi la notion de pipeline élémentaire englobe tous ces éléments qui seront à leur tour potentiellement répliqués par les outils OpenCL (détaillés dans le Tableau V.2).

III.2.3.2 Nombre de cycles

Il est possible que certaines données utilisées et/ou produites par le pipeline de calcul soient mises en cache dans des mémoires privées ou locales du FPGA, mais ce mécanisme, illustré à la Figure III.5, a pour conséquence d'augmenter la profondeur du pipeline, et donc d'en augmenter le temps d'établissement.

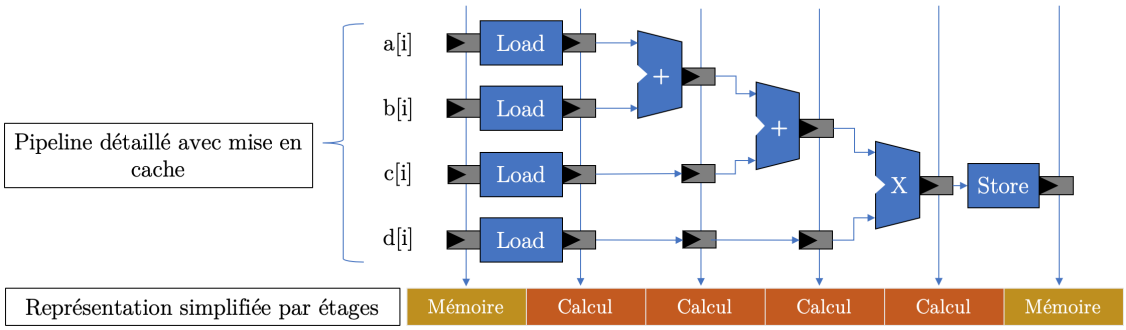


FIGURE III.5 – Pipeline avec mise en cache local

Ces nouvelles structures mémoires rajoutent donc un certain nombre de cycles d'horloge, qui dépendent de la zone mémoire choisie, et d'autres paramètres détaillés aux sections IV.3 et IV.2.

Au final, si l'on note P_{MEM} la profondeur correspondant aux étages d'accès mémoire, nous pouvons établir que la profondeur du pipeline élémentaire P_{PE} incluant les étapes de calculs et de mise en cache est donnée par l'équation (III.3).

$$P_{PE} = P_{PC} + P_{MEM} \tag{III.3}$$

Ainsi, le nombre de cycles d'horloge nécessaire pour exécuter tout le pipeline élémentaire correspond à la somme des cycles de calculs et des cycles dus aux mises en cache. Nous obtenons ainsi (III.4)

3. *local* ou *private*.

Nombre de cycles d'un pipeline élémentaire (noté $cycles_{PE}$) : c'est le nombre de cycles nécessaires au traitement de toutes les itérations par un pipeline élémentaire avec latences mémoires.

$$cycles_{PE} = P_{PC} + P_{MEM} + II * (iter_{PC} - 1) \quad \text{(III.4)}$$

III.2.4 Réplication du pipeline élémentaire - modèle *roofline*

Une fois le pipeline élémentaire construit, les outils d'analyse de Xilinx et d'Intel permettent de prédire les ressources et la bande passante que vont utiliser ce pipeline. A partir de ces caractéristiques, nous allons pouvoir estimer son facteur de réplication maximal, à l'aide d'un principe représenté sous forme graphique, connu sous le nom de *Roofline Model* [Williams et al., 2009].

Le but de ce modèle est de fournir une estimation de performances en considérant les limitations en puissance de calcul et en bande passante d'une architecture donnée.

Un algorithme peut être limité par ses accès mémoires (*memory bound*) ou bien par ses calculs (*compute bound*), et il existe différentes métriques permettant de le classifier rapidement dans l'une ou l'autre des catégories.

Nous retiendrons en particulier l'**intensité arithmétique**, dont la formule est donnée en (III.5).

Intensité Arithmétique(notée I_A) : elle correspond au ratio entre le nombre d'opérations à effectuer et le nombre d'accès mémoires de l'algorithme (en entrée et en sortie).

$$I_A = \frac{\#Operations}{\#Memory} \quad \text{(III.5)}$$

Ainsi, plus I_A est grand, plus l'algorithme correspondant sera limité par les calculs. À l'inverse, plus il est petit, plus il le sera par les accès mémoire.

Le modèle *roofline* est en réalité une méthode visuelle qui permet d'observer rapidement les possibles limitations de notre algorithme par rapport aux performances maximales théoriques sur l'architecture cible. Ainsi, un modèle est spécifique à une carte précise, mais son principe peut lui être décliné en de nombreuses variantes. Nous allons ici en présenter quelques-unes, puis introduire une adaptation de cette modélisation aux plateformes FPGAs.

III.2.4.1 Roofline simplifié

Afin de caractériser grossièrement les performances d'une architecture donnée, il suffit de récupérer :

— BP : sa **bande passante** maximale d'entrées/sorties en Bytes par seconde,

— P_{MAX} : sa **performance** maximale en opérations par seconde.

A partir de ces données, nous pouvons construire le graphique représentant le modèle, comme illustré à la Figure III.6.

Pour un algorithme donné, dont nous pouvons calculer l'intensité arithmétique, il est alors possible d'obtenir la performance maximale atteignable sur l'architecture considérée.

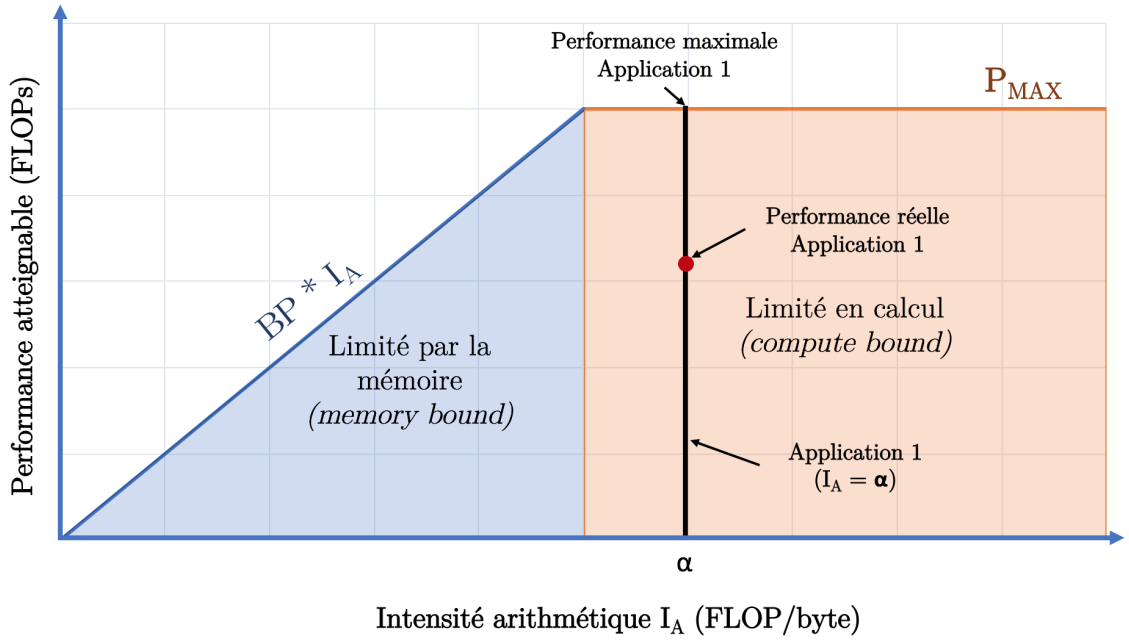


FIGURE III.6 – Caractérisation d'une application sur un Roofline Naïf

La fonction qui borne le domaine des performances atteignables est donnée par l'équation (III.6).

$$Roofline_{simple} = min(P_{MAX}, BP * I_A) \tag{III.6}$$

avec $Roofline_{simple}$ en FLOPS,
P_{MAX} en FLOPS,
BP en Bytes/s,
I_A en FLOP/Bytes

Une fois cette caractérisation de l'architecture effectuée, il s'agit alors, pour chaque algorithme, de calculer son intensité arithmétique, puis de tracer la droite verticale correspondante sur le graphe.

Il est alors possible de récupérer deux informations :

— si l'algorithme est limité par la mémoire ou par les calculs sur l'architecture donnée,

— la performance maximale atteignable avec cette intensité arithmétique.

Dans l'exemple de la Figure III.6, l'application 1 est limitée en calcul, et sa performance théorique maximale est égale à la performance maximale atteignable sur l'architecture considérée.

En ajoutant sur le graphique la performance réelle de l'application, le programmeur a une idée du potentiel restant d'accélération de son algorithme.

III.2.4.2 Roofline étendu

Dans la plupart des architectures modernes, le modèle *roofline* atteint rapidement ses limites. En effet, il existe souvent plusieurs types de mémoires, et refléter en une courbe la bande passante de l'architecture implique de n'en choisir qu'une.

De plus, les performances maximales sont souvent atteintes en exploitant toutes les optimisations possibles sur l'architecture donnée, et il est possible, en reprenant le principe du modèle *roofline* simple, de le compléter en faisant apparaître d'autres paramètres, comme les différents types de caches, ou encore le parallélisme d'instruction et la vectorisation présents dans les CPUs classiques.

Cela nous donne alors alors un graphique plus détaillé, similaire à celui de la Figure III.7.

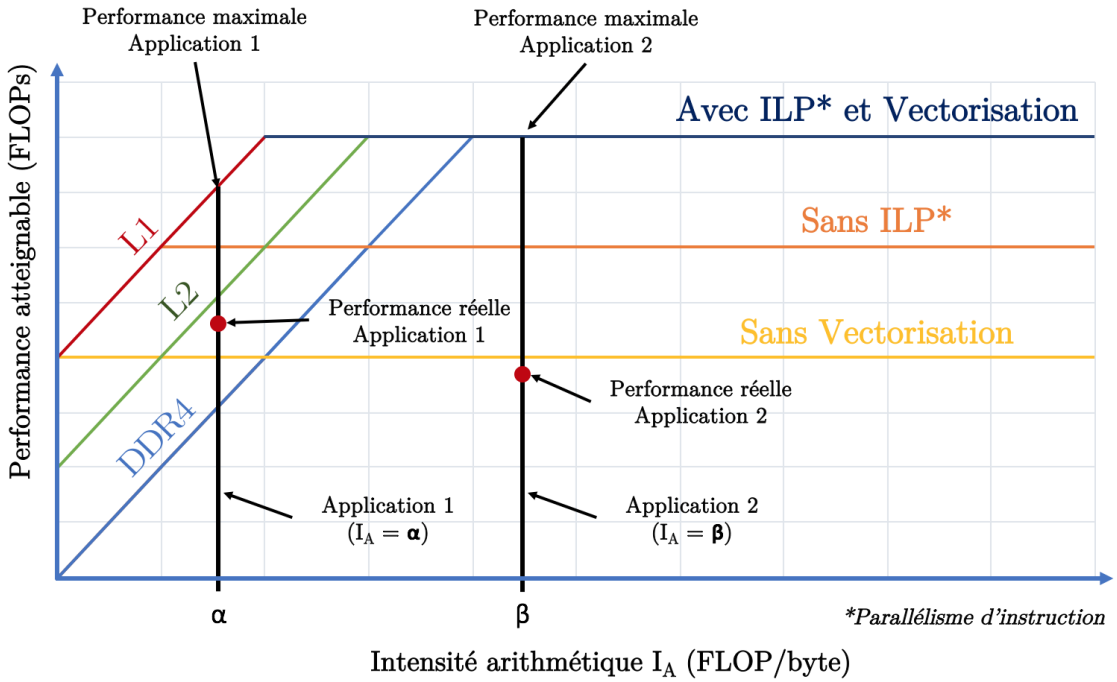


FIGURE III.7 – Caractérisation d'applications sur un Roofline étendu avec caches

Considérons les deux applications de cette figure. Par lecture graphique, nous pouvons savoir que :

— l'application 1, limitée en mémoire, tire déjà partie du cache L2. De plus, elle utilise ou la vectorisation, ou le parallélisme d'instructions. Les pistes possibles pour en améliorer la performance sont soit d'identifier des objets mémoires pouvant être mis

en cache L1, soit d'utiliser pleinement le parallélisme et la vectorisation d'instructions.

— l'application 2, limitée par les calculs, doit tirer parti du parallélisme d'instruction et de la vectorisation afin d'obtenir de meilleures performances

La réelle difficulté finalement, n'est pas de faire une lecture graphique pour une application précise, mais bien d'arriver à dresser un modèle *roofline* fiable de la plateforme considérée.

Sur une architecture matérielle figée, comme les CPUs et les GPUs, les débits correspondants aux différents seuils mémoires, ou les performances théoriques maximales en utilisant par exemple la vectorisation sont quantifiables, ce qui permet de pouvoir relativement facilement tracer les courbes correspondantes.

Par contre, sur une plateforme reprogrammable tel que les FPGAs, construire un tel graphique s'avère plus complexe, tant les possibilités sont variées. Toutefois, nous nous sommes proposés d'essayer d'adapter cette modélisation sur une architecture type de FPGA. Cette adaptation est présentée à la sous-section suivante.

III.2.4.3 Notre application du modèle *roofline* aux FPGAs

III.2.4.3.a Reprise d'un modèle existant

Différents travaux ont déjà vu le jour pour adapter le modèle *Roofline* aux FPGAs.

Les plus aboutis, [Silva et al., 2013] et [Yali, 2014] partent du constat que sur un FPGA, les performances dépendent tout autant de la plateforme cible que de l'algorithme à utiliser.

Leur méthodologie est la suivante :

— présenter les principales caractéristiques des FPGAs qui pourraient permettre d'adapter le modèle *Roofline* à ces plateformes,

— assimiler le parallélisme sur FPGA au déroulage de boucle,

— observer les performances et l'utilisation logique de différentes implémentations de deux algorithmes avec des facteurs différents de déroulage de boucle,

— en déduire une extrapolation du modèle sur les FPGAs.

Même si ces travaux sont complets et bien détaillés, leur approche focalisée sur le déroulage de boucle est limitant, en cela qu'il existe de nombreuses autres méthodes pour exprimer les multiples formes de parallélismes sur les FPGAs avec les outils OpenCL, comme par exemple la vectorisation, les boucles en pipeline, la réplication des kernels élémentaires, la démultiplication des piles d'instructions, ou encore les NDRKs (ces optimisations sont présentées au cours de ce chapitre). Ne prendre en considération que le déroulage de boucle est alors trop réducteur, et n'est adapté qu'au sous-groupe d'applications pour lequel cette solution est la plus optimale. En réalité, cette réduction en représentativité dénote de la difficulté de transposer aux FPGAs un modèle trop précis.

Ainsi, pour éviter l'approche réductrice évoquée précédemment, et pour ne pas se restreindre à un type de parallélisme ou à une classe d'algorithme trop peu représentatifs, nous allons dresser la liste des domaines d'optimisations que nous considérons

pertinents, pour ensuite effectuer une approche itérative d'optimisation qui sera illustrée à la section V.2 en fonction des spécificités des implémentations obtenues.

Notre approche du Roofline :

Parce que l'essence du modèle *roofline* est de pouvoir fournir une représentation synthétique rapide des limites des performances possibles d'une application sur une plateforme, nous avons proposé la première étape est d'identifier les éléments permettant de retrouver comme dans le modèle original des bornes supérieures de performance.

Les éléments reprogrammables sont majoritairement dédiés aux calculs, sauf cas spécifiques. Ainsi, il est possible de calculer précisément la bande passante entre l'hôte et le FPGA, ou entre les cœurs de calcul et la mémoire globale d'un FPGA donné.

Par contre, pour les blocs dédiés à l'implémentation des calculs (LUTs et DSPs) sur une puce donnée, le défi est de pouvoir caractériser leur performance maximale théorique. Or, cette valeur est, d'une part, très complexe à calculer, et d'autre part, n'a pas d'intérêt car impossible à retrouver dans un algorithme classique.

Ainsi, pour traduire le plus précisément possible la spécificité des FPGAs, nous nous inspirons du modèle *roofline* précédent et définissons un nouveau paramètre, $replication_{factor}$ détaillé plus loin en (III.9). Ce facteur de réplication permet de traduire, à partir d'une première version de notre algorithme, la réplication potentielle de ce dernier sur une architecture précise, en fonction de la consommation en ressources et de la bande passante par rapport aux valeurs maximales disponibles.

Sachant que l'implémentation d'un algorithme (Pipeline Élémentaire, PE) va utiliser un certain nombre de blocs (FF, LUT,DSP, BRAM), en comparant leurs utilisations au nombre maximal de ressources disponibles, nous avons fait le choix de définir le paramètre empirique, $ressource_{factor}$, détaillé en (III.7).

$$ressource_{factor} = min(\frac{FF_{FPGA}}{FF_{PE}}, \frac{LUT_{FPGA}}{LUT_{PE}}, \frac{DSP_{FPGA}}{DSP_{PE}}, \frac{BRAM_{FPGA}}{BRAM_{PE}}) \quad \text{(III.7)}$$

Ce paramètre est un indicateur des ressources utilisées empiriques par l'implémentation d'un algorithme, et représente donc le facteur de réplication lié aux ressources.

En ce qui concerne les différentes bandes passantes, il est possible, comme lors de la définition du modèle *roofline*, d'obtenir les valeurs correspondantes à partir des fiches techniques.

Comme nous nous intéressons ici à la réplication d'un algorithme élémentaire, la bande passante qui nous intéresse est la zone mémoire la plus limitante vue depuis la puce FPGA, c'est à dire la mémoire globale du FPGA, que ce soit de la mémoire DDR4 ou HBM2 par exemple.

A partir de ces informations, nous pouvons calculer le facteur de réplication lié à la bande passante (III.8), qui correspond au ratio entre la bande passante d'un pipeline élémentaire et la bande passante disponible sur l'architecture.

$$bandwith_{factor} = \frac{bandwith_{FPGA}}{bandwith_{PE}} \quad \text{(III.8)}$$

En mettant en commun les contraintes en ressources et en bande passante, nous définissons le $replication_{factor}$, détaillé par la formule III.9.

Facteur de réplication $replication_{factor}$ **:** ce paramètre correspond au facteur maximal de réplication de notre pipeline élémentaire sur une architecture précise.

$$replication_{factor} = min(bandwith_{factor}, ressource_{factor}) \quad \text{(III.9)}$$

De concert avec le modèle *roofline*, ce paramètre est un indicateur des performances atteignables si l'algorithme considéré peut être répliqué.

III.2.4.3.b Limites de cette approche

Il convient encore une fois de rappeler ici que ce modèle est un modèle empirique, qui n'a d'intérêt que couplé à une analyse pertinente des résultats.

Ainsi, il s'agit de repérer rapidement les pistes d'optimisations possibles qui pourront permettre, probablement, d'améliorer l'implémentation d'un algorithme.

La principale limite vient de l'absence d'indicateur de performance maximale pour les LUTs. Le paramètre de réplication introduit permet dans une certaine mesure d'ajouter une contrainte liée à ces blocs de calcul, mais reste dépendant de l'efficacité de l'outil dans la traduction d'un algorithme en blocs matériels.

Une seconde limite importante est que d'une part l'optimisation efficace d'un algorithme sur FPGA passe par l'utilisation de la mémoire locale et privée disponible sur la puce même, et que d'autre part, parvenir à mesurer la bande passante maximum de ce type de mémoire est impossible car dépendant de l'implémentation donnée pour chaque algorithme.

III.3 Modèle proposé de prédiction du temps d'exécution d'une application sur FPGA

A partir des notations et concepts énoncés, nous nous sommes proposés de prévoir le temps d'exécution d'une application à partir des paramètres récupérés dans les outils d'analyse statique des différents ateliers FPGA.

Des exemples observés, nous avons déduit, en supposant une utilisation efficace des copies entre l'hôte et la plateforme d'accélération, une modélisation du temps théorique d'exécution $T_{theorique}$ qu'exprime l'équation (III.10) (Notations : Tableau III.1)

Temps d'exécution modélisé d'une application FPGA :

$$T_{theorique} = \frac{P_{PC} + P_{MEM} + II * (\frac{iter_{PC}}{replication_{factor}} - 1)}{f_{kernel}} \quad \text{(III.10)}$$

Cette modélisation simplifiée permet de souligner l'importance de plusieurs facteurs. L'intervalle d'initialisation peut très rapidement , de même qu'une fréquence basse, augmenter le temps d'exécution d'une application. L'utilisation de la mémoire locale a un

avantage indéniable en cela que l'augmentation de la profondeur du pipeline est souvent négligeable par rapport aux nombre de cycles des boucles.

Ces observations expliquent en partie la stratégie adoptée par les constructeurs dans leurs outils, qui se concentrent en priorité sur :

1. réduire au maximum l'intervalle d'initialisation des boucles,
2. obtenir une fréquence la plus élevée possible

Toutefois, il est à noter que l'intervalle d'initialisation projeté par les outils peut ne pas être respecté au moment de l'exécution. En effet, si par exemple, dans le corps de la boucle, il y a un accès en mémoire globale par itération, il est possible, suivant les défauts ou succès de cache, que certaines itérations aient un temps de blocage non prédictible à la compilation qui se répercute sur le temps théorique précédent.

Il est donc important d'utiliser la formule III.10 comme un indicateur de performance qu'il faut savoir nuancer suivant la complexité du code et des accès mémoires.

Chapitre IV

Optimisations OpenCL proposées

Sommaire

Ce chapitre s'appuie en partie sur les différents concepts énoncés précédemment, notamment ceux de la section III.2, ainsi que sur certaines optimisations présentées par les constructeurs et intégrés dans leurs outils.

Nous établissons, dans cette section, un manuel d'optimisation des FPGAs avec OpenCL. Nous y avons intégré d'une part les concepts d'optimisations avancées que nous avons pu définir, et d'autre part les optimisations proposées par les constructeurs qui nous ont parues les plus intéressantes, pour lesquelles nous avons défini les conditions dans lesquelles leur utilisation est ou non avantageuse.

Remarques préliminaires :
Pour la clarté du manuscrit, il convient ici de préciser deux notations adoptées dans cette section.
1. Il existe deux catégories de kernels en OpenCL, comme présenté à la section II.6. On retrouve donc les Single Work-Item Kernels (SWIKs) et les NDRange Kernels (NDRKs), qui, de par leurs caractéristiques, ont dans certains cas des optimisations spécifiques. Ainsi, au cours de cette section, une sous-section qui contient SWIK ou NDRK dans son titre n'est applicable qu'à la catégorie de kernel correspondante. En l'absence de l'un ou de l'autre mot clef, l'optimisation est valable quelque soit le type de kernel.
2. Les travaux d'optimisation sur les FPGAs en OpenCL se sont focalisés sur deux ateliers de développement en parallèle, à savoir SDAccel associé à Vivado pour Xilinx, et le *Intel FPGA SDK for OpenCL* associé à Quartus pour Intel. Ainsi, comme pour la remarque précédente, la présence d'un mot clef Intel ou Xilinx dans le titre d'une d'optimisation la rend exclusive à cette plateforme. L'absence de ces mots clefs signifie donc que la catégorie correspondante est compatible avec les deux outils et constructeurs.

IV.1 Optimisation du pipeline de calcul

IV.1.1 Représentation des données et opérations

IV.1.1.1 Types

Bien que, dans la plupart des cas, l'utilisation de la virgule flottante ne soit pas nécessaire, et qu'elle puisse facilement être remplacée par une arithmétique en virgule fixe, son intérêt majeur est de pouvoir adapter dynamiquement la précision des nombres de façon à représenter une plage de réels plus importante. Cet avantage explique en partie pourquoi la quasi totalité des CPUs depuis le processeur Intel 80486 [1] intègrent des unités de calcul en virgule flottante.

Toutefois, cette polyvalence n'est pas sans coût, puisque le stockage de l'exposant nécessite l'ajout de ressources matérielles et augmente également la latence des opérations.

sur FPGA, il est possible d'utiliser différents types de représentation suivant le besoin du programme. Dans le cas où l'utilisation de la virgule flottante à double précision ou "half precision" est nécessaire, il est toujours possible, en incluant respectivement les

1. Commercialisé en 1989.

directives (IV.1) et (IV.2) d'ajouter la prise en charge de ces types par les outils OpenCL correspondants.

$$\#pragma\ OPENCL\ EXTENSION\ cl\_khr\_fp64\ :\ enable \tag{IV.1}$$

$$\#pragma\ OPENCL\ EXTENSION\ cl\_khr\_fp16\ :\ enable \tag{IV.2}$$

Motivations :

— Maximiser les performances d'une application en co-processing passe par l'optimisation de l'application sur l'accélérateur, mais aussi par une gestion efficace des communications entre les différentes plateformes. Ce qui suppose une optimisation de la gestion de la mémoire. La particularité d'OpenCL est de garantir à chaque type de donnée, la même empreinte mémoire, le même alignement en nombre de Bytes quelque soit l'architecture cible.

— Les FPGAs ont comme particularité de pouvoir gérer les différentes représentations des nombres, et il est souvent utile d'analyser les données manipulées afin d'éventuellement les transformer en représentation moins coûteuses en ressources, tout en s'assurant que les calculs finaux répondent toujours aux spécifications établies.

Le tableau IV.1 dresse la correspondance entre chaque type et son alignement effectif en OpenCL.

TABLE IV.1 – Taille réelle en Octets des types courants en OpenCL

Type (host)	**Type (FPGA)**	**Alignement (Octets)**
cl_char, cl_uchar	char, uchar	1
cl_short, cl_ushort, cl_half	short, ushort, half	2
cl_int, cl_uint, cl_float	int, uint, float	4
cl_long, cl_ulong, cl_double	long, ulong, double	8
–	long long, ulong long	16
Types vectorisés : n = 1,2,4,8,16		
cl_type*n*	type*n*	n * sizeof(type)
Types vectorisés (exception) : n = 3		
cl_type*3*	type*3*	4 * sizeof(type)

Le cas des types scalaires ne pose aucune difficulté, seuls les types vectorisés de facteur 3 ont un alignement particulier. Pour la très grande majorité des architectures, les tailles de bus ou de structures mémoires, ne sont pas divisible par 3, la gestion d'une donnée dont la taille en mémoire est un multiple de trois s'avère donc complexe, du fait de la systématique sous-utilisation des différents mécanismes mémoires, tant pour la copie que pour le stockage.

Aussi, le choix d'OpenCL a été d'aligner tous les types vectorisés de trois termes sur la structure correspondante à 4 termes. Un vecteur de trois entiers sera donc aligné sur $4 * sizeof(int) = 16\ Bytes$ au lieu de $3 * sizeof(int) = 12\ Bytes$.

L'utilisation des vecteurs de longueur trois est toujours possible, mais n'a d'attrait qu'eu égard aux considérations de compréhension. Dans la manipulation des vecteurs en géométrie 3D, l'utilisation de tels vecteurs est conceptuellement logique, mais il est préférable

et recommandé par le standard OpenCL d'utiliser toutefois des vecteurs de longueur 4, quitte à ce que la dernière coordonnée soit inutilisée.

Conditions appropriées d'utilisation :

— utiliser des vecteurs de longueur 4, pour la manipulation des vecteurs 3D

— maximiser la vectorisation en fonction de la taille des bus de l'architecture cible

— vérifier l'alignement adéquat des types de données, en particulier quand ils sont regroupés au sein d'une même structure.

IV.1.1.2 Structures

OpenCL permet la définition de structures, à l'aide de n'importe quelle combinaison des types de base.

Toutefois, les compilateurs OpenCL gèrent de différentes manières la définition de ces structures, notamment en ce qui concerne leur alignement en mémoire. Prenons l'exemple en C, de la définition d'une structure mémoire du côté de l'hôte (Algo. 2), et celle correspondante du côté du kernel (Algo. 3).

Algorithme 2 : Exemple de structure (host)

```
1 typedef struct Host {
2     cl_int x ; /* 4 Bytes */
3     cl_int y ; /* 4 Bytes */
4     cl_int2 z ; /* 8 Bytes */
5 } tStructHost;
```

Algorithme 3 : Correspondance (kernel)

```
1 typedef struct Kernel {
2     int x;
3     int y;
4     int2 z;
5 } tStructKernel;
```

Dans le cas de ces structures, un int est aligné sur 4 Bytes, et un $int2$ l'est sur 8 Bytes. Pourtant, une structure va aligner tous ses membres sur des multiples de l'alignement du membre le plus long. La structure aura en réalité tous ses membres alignés sur 8 Bytes, et non sur 4 pour les entiers.

La taille minimale finale de la structure sera donc de $8 * 3 = 24$ Bytes et non de $4 + 4 + 8 = 16$ Bytes

La définition de structures est possible, mais il faut donc prendre en compte les mécanismes d'alignement inclus dans les compilateurs OpenCL pour FPGA.

Conditions appropriées d'utilisation :

— Pour l'utilisation des types vectorisés du côté de l'hôte, il est nécessaire d'ajouter le préfixe $cl\_$ devant leurs éventuelles déclarations. Il est donc recommandé dans ce cas d'avoir deux définitions de structures différentes, l'une pour l'hôte, l'autre pour le kernel, comme illustré précédemment.

— L'alignement des types scalaires et vecteurs se fait suivant la table IV.1.

— Une structure a pour alignement celui de son membre d'alignement maximum.

IV.1.1.3 Opérations

En pratique, un FPGA peut effectuer toutes les opérations possibles sur les CPUs ou GPUs. Toutefois, certaines d'entre elles ont une implémentation coûteuse en ressources et peu efficace, et il peut être intéressant de bien identifier ces configurations pour éventuellement déporter les calculs correspondants sur l'hôte.

Conditions appropriées d'utilisation :

La flexibilité permise par le principe du co-processing peut être, par exemple, utilisée dans le cas suivant. Si, dans un kernel FPGA, nous avons une constante dont le calcul est réputé être peu efficace sur FPGA, alors il est possible de la pré-calculer sur l'hôte, et de transférer le résultat au kernel. Dans ce cas, il faut bien veiller à ce que le temps de copie et de calcul sur CPU ne dépasse pas le temps de calcul initial sur FPGA.

Toutefois, un certain nombre d'opérations standard définies par OpenCL sont prises en charge par les outils FPGA. L'utilisation d'une fonction native (préfixe $native\_$) prise en charge par l'outil garantit une implémentation efficace de la fonction correspondante, car optimisée par le constructeur pour l'architecture précise, comme par exemple les fonctions $native\_cosinus$, $native\_sinus$, $native\_sqrt$ (fonction racine carrée) et $native\_pow$ (fonction puissance).

De manière similaire, si une fonction à implémenter existe déjà dans une librairie fournie, il est souvent préférable de l'utiliser.

IV.1.2 Cas des boucles

L'optimisation des boucles sur FPGAs est l'étape la plus cruciale, en cela qu'elles permettent l'expression des différentes formes de parallélisme d'un programme.

Néanmoins, un développeur voulant optimiser un code sur un FPGA via OpenCL se doit d'avoir à l'esprit les spécificités de cette architecture, notamment le fait que l'écriture d'un algorithme avec une variable de fin de boucle définie de manière dynamique est souvent synonyme de mauvaises performances.

En effet, avec les outils de Xilinx, paralléliser une boucle simple dont la variable de fin de boucle est définie à l'exécution conduit à une génération incorrecte de l'architecture correspondante. Il faut alors réécrire cette boucle simple en la divisant en deux parties, la première dont le nombre d'itérations divise le facteur de réplication, et une boucle d'épilogue qui gère les dernières itérations. Cela demande toutefois une réécriture du code substantielle, qui, si elle n'est pas faite, entraîne ces mauvaises performances.

Cette sous-section a pour vocation à illustrer d'une part certaines optimisations connues, tout en en montrant leurs limites, mais également à définir de nouvelles optimisations construites à partir des briques élémentaires fournies par les constructeurs.

Un certain nombre d'optimisations basiques ne seront ici pas évoquées, et on laissera au lecteur le soin de se référer aux documentations techniques [Xilinx, 2018] [Int, 2017] des outils correspondants.

IV.1.2.1 Pipeline d'une boucle

L'optimisation fondamentale, lorsque l'on s'intéresse aux boucles d'un programme ciblant les FPGAs, est la transformation en pipeline d'une boucle. Illustré à la Figure IV.1,

le principe est de segmenter en une succession d'opérations le corps de la boucle, et d'alimenter en décalé chaque étage par de nouvelles itérations.

S'il n'y avait pas de pipeline de boucle, et en notant $cycles_{iter}$ le nombre de cycles que prend une itération, on aurait alors un nombre de cycles total minimum donné par (IV.3) pour traiter la boucle dans son intégralité.

$$cycles_{noPipeline} = iter_{boucle} * cycles_{iter} \tag{IV.3}$$

où $cycles_{iter}$ correspond au nombre de cycles pour traiter une itération,
$iter_{boucle}$ correspond au nombre d'itérations de la boucle.

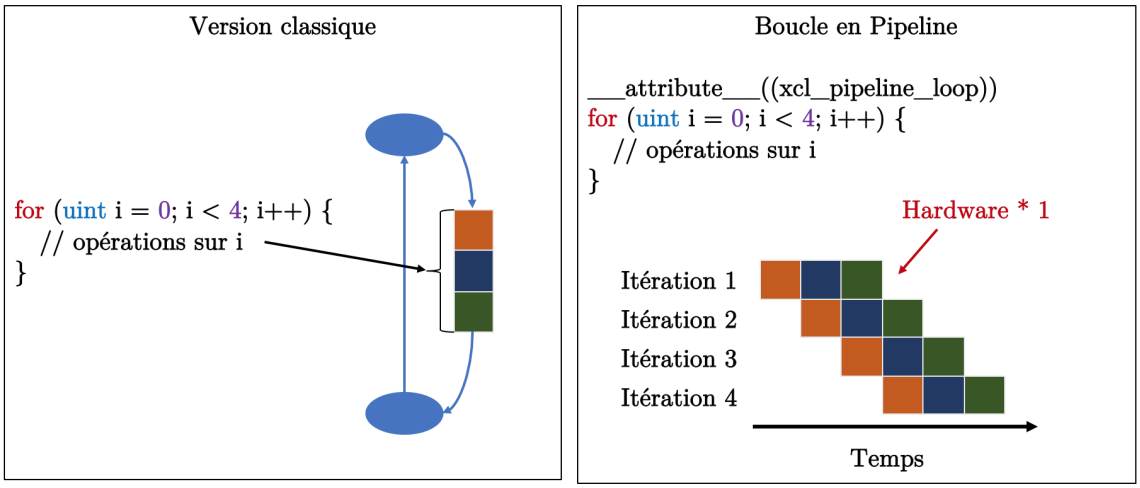


FIGURE IV.1 – Illustration d'une boucle en pipeline.

En transformant la boucle en pipeline, nous pouvons appliquer les notions précédemment définies à la section III.2.2.1, ce qui nous donne l'équation de calcul du nombre de cycles total minimum pour traiter l'intégralité des itérations (IV.4) :

$$cycles_{Pipeline} = iter_{boucle} * II + cycles_{iter} \tag{IV.4}$$

En terme de ratio performance/ressources utilisées, cette solution est la plus optimale. En effet, en plus de réduire le temps d'exécution total de la boucle, le matériel généré correspondant est mis en commun entre les itérations, et il n'y a qu'un léger surcoût pour le mécanisme d'alimentation des itérations.

Cette optimisation est à privilégier sur les FPGAs, et de nombreux mécanismes s'appuient sur cette notion.

IV.1.2.2 Déroulage de boucle

Néanmoins, une seconde optimisation est possible pour tirer parti du parallélisme des boucles. Il s'agit du déroulage de boucle, qui consiste à répliquer, avec un facteur paramétrable, le corps d'une boucle.

Dérouler une boucle, comme illustré à la Figure IV.2 signifie donc que les ressources utilisées pour traduire le code du corps de la boucle sera répliqué, y compris certaines variables mémoires. Aussi, en considérant uniquement les performances brutes, cette solution est intéressante puisqu'on augmente localement le parallélisme de l'algorithme, tout en ne répliquant pas entièrement le code.

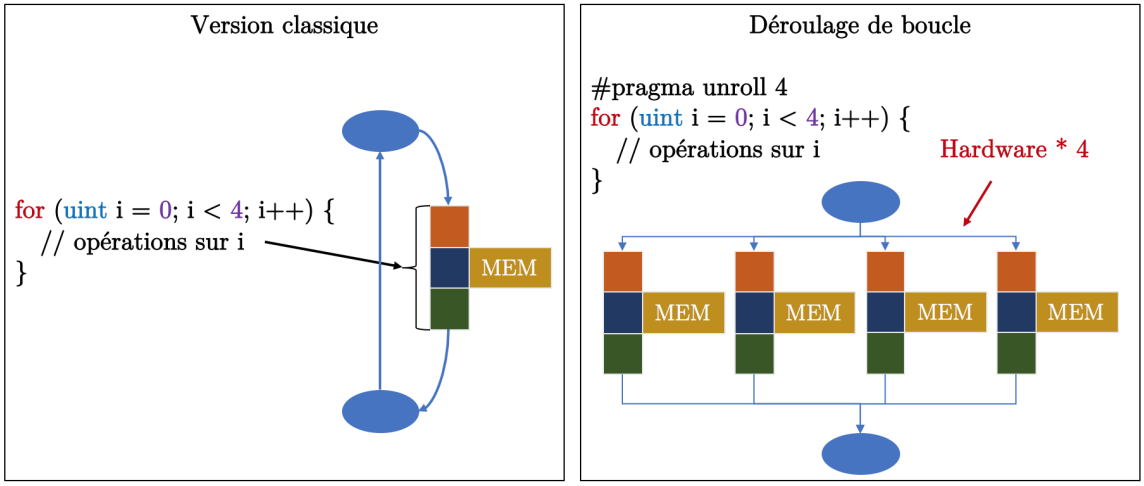


FIGURE IV.2 – Réplication du hardware correspondant lors d'un déroulage de boucle.

Toutefois, si la partition des objets mémoires a été mal pensée, le compilateur risque de répliquer des variables qui n'ont pas lieu de l'être, consommant inutilement des ressources matérielles et rendant la génération d'une architecture efficace plus ardue.

IV.1.2.3 Tests conditionnels et accès mémoire

Aussi, les compilateurs haut niveau essayent en priorité d'optimiser les boucles présentes à l'intérieur d'un code, et il est très pénalisant d'avoir des boucles à l'intérieur d'un branchement conditionnel. Il faut donc veiller, dans le mesure du possible, à insérer les branchements conditionnels à l'intérieur d'une boucle, et non l'inverse.

Plus généralement, quand on considère les accès mémoire, il faut veiller autant que possible à ne pas déséquilibrer les branchements conditionnels.

Considérons, à l'aide d'un exemple, les accès mémoires, et prenons l'algorithme 4 dont le but est de sommer une valeur sur deux d'un tableau de SIZE_ARRAY entiers.

Algorithme 4 : Somme d'un élément sur deux d'un tableau

```
1 int accum = 0 ;
2 for ( int i = 0 to SIZE_ARRAY - 1 ) {
3 |   if i%2 == 0 then
4 |   |   accum += tab[i];
```

Algorithme 5 : Optimisation correspondante

```
1 int accum = 0 ;
2 for ( int i = 0 to SIZE_ARRAY - 1 ) {
3 |   int is_good = 0;
4 |   if i%2 == 0 then
5 |   |   is_good = 1 ;
6 |   accum += tab[i] * is_good;
```

Cette première version entraîne un déséquilibre au niveau des branchements, et le compilateur aura alors des difficultés à prédire la régularité d'accès à notre tableau de départ.

Par contre, en le réécrivant, comme présenté à l'Algorithme 5, l'équilibre est rétabli. Certes, nous avons dorénavant un accès en mémoire pour chaque itération, contre une fois sur deux lors du premier algorithme, mais le compilateur peut alors inférer des caches efficaces grâce à la régularité des nouveaux accès mémoires.

Conditions appropriées d'utilisation :

— Il faut veiller, dans le mesure du possible, à insérer les branchements conditionnels à l'intérieur d'une boucle, et non l'inverse.

— Dans le cas où un accès mémoire est nécessaire dans le corps de la boucle, il faut veiller autant que possible à ne pas déséquilibrer son accès dans les branchements conditionnels.

IV.1.2.4 Boucles imbriquées

Remarque : *dans cette sous-section, on ne tient pas compte du temps d'établissement du pipeline car on le considère, dans notre cas de figure, négligeable devant le temps de traitement de toutes les itérations des boucles.*

Il n'est pas rare que l'on se retrouve avec différentes boucles imbriquées les unes dans les autres.

Comme illustré à la Figure IV.3, la gestion des itérations d'une boucle rajoute une latence, à cause notamment des tests de sortie de boucle et de l'incrémentation de l'itérateur.

```
for (int i = 0; i < A ; i++) { /* Boucle 1 */
  int new_i = i - off_i ;
  for (int j = 0; j < B ; j++) { /* Boucle 2 */
    int new_j = j - off_j ;
    for (int k = 0; k < C ; k++) { /* Boucle 3 */
      int new_k = k - off_k ;
      /* utilisation des new_i,j,k  (prend 29 cycles)*/

    }

  }

}
```

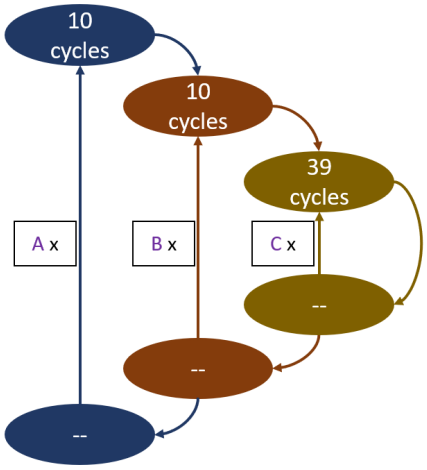


FIGURE IV.3 – Boucles imbriquées simples et nombres de cycles correspondant.

Dans le cas illustré, les boucles 1, 2, et 3 ont comme nombre de cycles respectifs 10, 10 et 39, c'est à dire qu'il faut par exemple 39 cycles pour traiter une itération de la boucle 3 (10 pour la gestion de la boucle, 29 pour le corps).

Ainsi, le nombre de cycles théorique pour traiter ce programme est donné en (IV.5), où A, B, et C représentent le nombre d'itérations respectives des boucles 1, 2, et 3.

$$cycles_{algo1} = A * (10 + B * (10 + 39 * C)) \tag{IV.5}$$

Pour réduire la propagation de la latence liée à la gestion des boucles, il est possible de demander, à l'aide de la directive (IV.6), au compilateur de fusionner les boucles, ce qui a pour conséquence de mettre en commun leur gestion.

L'argument *N* permet de dérouler les boucles suivant la profondeur *N* correspondante. Ainsi, si N = 2, seuls les deux premiers niveaux de boucle seront fusionnés, alors que si N = 3, les trois niveaux de boucle le seront.

$$\#pragma\ loop\_coalesce(N) \tag{IV.6}$$

Ajouter $\#pragma\ loop\_coalesce(3)$ comme illustré à la Figure IV.4 permet de paramétrer la fusion des trois boucles, ce qui entraîne bien la mise en commun de la gestion des itérateurs. Conceptuellement, ajouter cette fusion de boucles équivaut à réécrire l'algorithme comme celui que l'on retrouve à droite de cette figure.

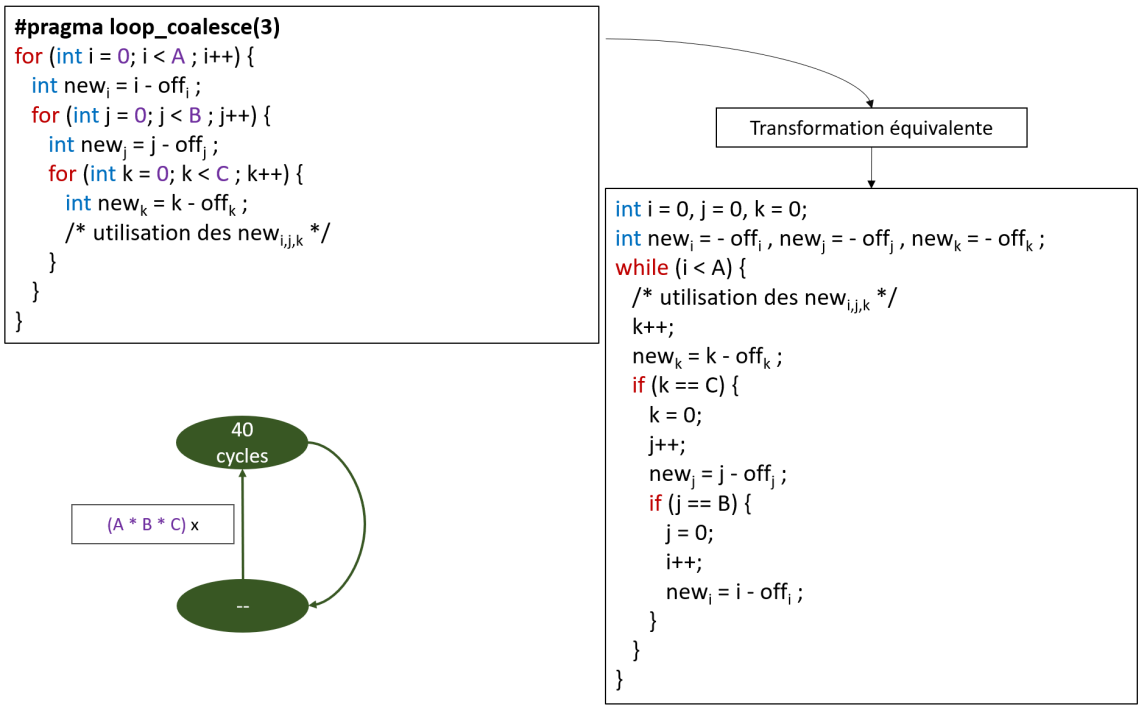


FIGURE IV.4 – Fusion de boucles imbriquées et nombres de cycles correspondant.

Néanmoins, si cette optimisation peut dans certains cas réduire le nombre de cycles totaux de notre algorithme, elle peut tout à fait en augmenter la latence, en particulier pour un grand nombre d'itérations.

En effet, le nombre de cycles théoriques de l'optimisation de la Figure IV.4 est donné

par la formule (IV.7).

$$cycles_{algo2} = A * B * C * 40 \tag{IV.7}$$

Or, si l'on suppose que $A, B, C \in \mathbb{N}^*$, pour que l'optimisation soit efficace, l'inéquation IV.8 doit être respectée.

$$cycles_{algo2} < cycles_{algo1} \tag{IV.8}$$

Dans notre cas de figure, pour A, B, C strictement positifs, cette inéquation est équivalente à $(C - 10) * B < 10$.

Comme B et C sont des entiers strictement positifs, on en déduit finalement que (IV.9) :

$$\begin{cases} C \in [1; 20[\\ C \in [1; 10] \implies B \in \mathbb{N}^* \\ C \in]10; 20[\implies B < \frac{10}{C-10} < 10 \end{cases} \tag{IV.9}$$

Conditions appropriées d'utilisation :

En généralisant le cas particulier présenté ci-dessus, on peut retenir qu'il existe deux cas de figures où la fusion des boucles est intéressante :

— quand B et C sont suffisamment petits

— dans le cas des boucles parfaites

Une série de boucles imbriquées est dite **parfaite** s'il n'existe aucune instruction entre chacun des différentes niveaux de boucles, c'est à dire, si l'on reprend l'exemple de la Figure IV.3, qu'il n'y ait aucune instruction aux lignes 2 et 4.

Dans ce cas, le nombre de cycles de la boucle la plus imbriquée est égal au nombre de cycles de la boucle fusionnée.

Alors, l'inéquation IV.8 a comme solution $S = (A, B, C) \in \mathbb{N}^*$

Il convient donc de prêter une attention particulière aux boucles imbriquées, pour éviter de modifier notre implémentation et d'influer de manière négative sur les performances en pensant réduire le coût lié à la gestion des itérateurs.

IV.1.2.5 Dépendances à l'intérieur d'une boucle

Prenons comme exemple l'algorithme qui permet de calculer les N premières valeurs de Fibonacci 6 dans sa version naïve.

Algorithme 6 : Calcul des N premiers entiers de Fibonacci - Naïf

Input : N, où $N \geqslant 2$
Output : Fibo[N]

```
1 Fibo[0] = 0;
2 Fibo[1] = 1;
3 for ( i = 2 to N - 1 ) {
4 |   Fibo[i] = Fibo[i - 1] + Fibo[i - 2];
```

Le compilateur, en appliquant le pipeline de boucle, va donc reconnaître que, pour chaque itération, il y a besoin de deux lectures, une opération, et une écriture.

Mais, comme l'itération i a besoin que les deux itérations $i-1$ et $i-2$ soient terminées avant de pouvoir les sommer, la boucle aura un intervalle d'initialisation $II = 4$, en partant du principe que les quatre opérations prennent un cycle d'horloge.

Cet algorithme est inefficace au niveau du parallélisme, mais également parce que les données sont souvent réutilisées sans être mises en cache.

Il convient donc, dans le cas où le rapport de compilation nous indique que la contrainte de $II = 1$ n'a pas été respectée, de s'assurer qu'il n'y a pas de dépendance de boucle, et, si c'est le cas, de les limiter voire de les supprimer dans la mesure du possible.

Par exemple, il s'agirait pour la suite de Fibonacci d'appliquer les méthodes de calcul des termes d'une suite, ce qui nous permet, avec le nombre d'or, d'obtenir directement à l'aide d'un calcul de puissance les termes de la suite sans dépendance avec leurs prédécesseurs.

Conditions appropriées d'utilisation :

— Identifier, dans le cas où le compilateur n'arrive pas à garantir un intervalle d'initialisation de 1, s'il est possible de simplifier voire de supprimer les dépendances de boucle.

— Si la dépendance est liée à une variable d'accumulation, se référer à l'optimisation suivante.

IV.1.2.6 Accumulateurs (SWIK, Intel)

De nombreux algorithmes nécessitent l'utilisation d'un accumulateur à l'intérieur de boucles. Or, l'utilisation de ces derniers posent un certain nombre de problèmes, notamment en ce qui concerne les dépendances mémoires. Cela est d'autant plus vrai dans le cas où les boucles sont en pipeline ou déroulées, car il faut alors gérer les conflits d'accès à cet accumulateur.

Une spécificité d'Intel a été, depuis les cartes basées sur l'Arria 10, d'intégrer des mécanismes qui permettent de repérer une variable d'accumulation et de l'implémenter efficacement dans un pipeline. Pour cela, il suffit de déclarer et d'utiliser cette variable en suivant les recommandations du guide d'utilisation du compilateur d'OpenCL d'Intel.

	Pipelined	II	Bottleneck
Block3 (backprojection3D_kernel.cl:36)	Yes	1	n/a
Block4 (backprojection3D_kernel.cl:37)	Yes	1	n/a
Block5 (backprojection3D_kernel.cl:39)	Yes	1	n/a
4X Partially unrolled Block6 (backprojection3D_kernel.cl:50)	Yes	6	II

(a) DE1-SoC

	Pipelined	II	Bottleneck
backprojection3D.B1 (backprojection3D_kernel.cl:36)	Yes	>=1	n/a
backprojection3D.B2 (backprojection3D_kernel...	Yes	>=1	n/a
backprojection3D.B3 (backprojection3D_ke...	Yes	>=1	n/a
backprojection3D.B4 (backprojection3...	Yes	~1	n/a

(b) Arria 10

FIGURE IV.5 – Rapports de compilation d'un même code sur différentes technologies.

En Figure IV.5 sont illustrés les rapports de compilation d'un même code sur deux cartes différentes, celle de gauche étant un FPGA SoC basé sur un Cyclone V (DE1-SoC), celle de droite étant un FPGA PCIe basé sur un Arria 10.

Pour un code identique, alors que le design sur la DE1-SoC ne peut pas réduire l'intervalle d'initialisation de la dernière boucle imbriquée en dessous de 6 à cause de la dépendance en données d'un accumulateur, le même design généré pour l'Arria 10 permet d'obtenir un intervalle d'initialisation optimal de 1, notamment grâce aux derniers DSPs intégrés sur ces cartes, qui permettent d'effectuer une addition ou une soustraction flottante en un seul cycle d'horloge.

IV.1.3 Types de kernel

En section II.6, nous avons déjà évoqué les différents types de kernels : les Single Work Item Kernels (SWIKs) et les NDRangeKernel (NDRKs).

Pour la première catégorie, le parallélisme est à extraire dans le corps de la fonction, alors que pour la seconde catégorie, le parallélisme principal vient de l'expression du parallélisme de thread SIMT, similaire à ce que l'on retrouve sur les GPUs. Son principe est basé sur la discrétisation de l'espace du problème en une ou plusieurs dimensions, pour ensuite répartir sur un ou plusieurs "cœurs" de calcul la charge de travail.

Il est à noter qu'une des forces d'OpenCL, et particulièrement sur les FPGAs, est de pouvoir exprimer tous les types de parallélismes dans une même implémentation.

Les documentations techniques des constructeurs développent suffisamment les concepts liés à ces deux types de kernel pour que nous ne les reprenions ici. Toutefois, il convient de préciser que les barrières de synchronisation, ou encore les opérations atomiques sont, pour les NDRKs, des restrictions fortes pouvant réduire drastiquement les performances d'une application.

Conditions appropriées d'utilisation : Tant Xilinx qu'Intel préconisent de favoriser l'approche basée sur les SWIKs, et d'exprimer le parallélisme de données éventuel à l'aide du déroulage et du pipeline de boucle.

IV.1.4 Fréquence et intervalle d'initialisation

L'intérêt de réduire au maximum l'intervalle d'initialisation a déjà été discuté à la section III.3, mais il existe néanmoins certains cas où il est intéressant de ne pas imposer $II = 1$.

En effet, quand le compilateur OpenCL d'Intel analyse un code, il va vouloir en priorité obtenir **II = 1**. Mais, il ne peut y arriver certaines fois qu'en descendant la fréquence globale du kernel, et donc en ralentissant ce dernier d'autant.

Prenons comme exemple l'Algorithme 7 décrit ci-dessous, et supposons que :

- le temps d'exécution de la boucle 1 est relativement négligeable devant celui de la boucle 2,
- les boucles 1 et 2 sont bien en pipeline,
- le compilateur n'arrive à obtenir $II = 1$ pour la boucle 1 qu'en descendant la fréquence du kernel.

Dans ce cas, le tableau IV.2 permet de se rendre compte de l'incidence d'une augmentation manuelle du **II** de la première boucle sur le temps d'exécution global du kernel. Dans le premier cas de figure, on laisse le compilateur optimiser le code à sa façon. Il va donc tenter de pipeliner les deux boucles, et d'obtenir un intervalle d'initialisation qui

Algorithme 7 : Algorithme d'illustration de l'optimisation du **II**

1 Hypothèse : $T_{Boucle1} << T_{Boucle2}$

2 **for (** *i = 0 to A - 1* **)** {
3 | /* Calculs avec dépendance entre les boucles */
4 **for (** *j = 0 to B - 1* **)** {
5 | /* Longue boucle */

soit égal à 1 pour les deux boucles. Nous avons pris comme hypothèse qu'il n'arrivait à obtenir **II = 1** pour la première boucle qu'en échange d'une fréquence réduite pour tout le kernel.

En appliquant la modélisation du temps d'exécution présentée à la section III.2.2.3, on obtient que le temps d'exécution de la boucle 1 est plus rapide dans le premier cas de figure. Mais, comme cela entraîne une réduction de la fréquence globale de tout le kernel, cela ralentit également la seconde boucle, qui est donc défavorisée dans ce premier cas de figure.

Imposer **II = 2** pour la première boucle permet de ne plus avoir à réduire la fréquence générale, et, bien que cela augmente le temps d'exécution de la boucle 1, le gain est significatif pour la seconde boucle.

TABLE IV.2 – Optimisation d'un code par augmentation locale de l'intervalle d'initialisation.

	Cas de figure 1	**Cas de figure 2**
Boucle 1		
Itérations	A = 32	A = 32
Profondeur	20	20
II	1	2
Temps d'exécution (us)	0,46	0,53
Boucle 2		
Itérations	B = 1024	B = 1024
Profondeur	264	264
II	1	1
Temps d'exécution (us)	11,5	8,05
Boucles 1 & 2		
Fréquence Kernel	112 MHz	160 MHz
Temps d'exécution total (us)	11,96	8,58

A noter qu'il est possible également, afin de réduire les efforts nécessaires à l'optimisation d'un kernel, de manuellement préciser la fréquence maximum voulue pour un kernel, avec la directive (IV.10) au moment de la définition du kernel correspondant.

$$\_\_attribute\_\_((scheduler\_target\_fmax\_mhz(value))) \tag{IV.10}$$

Le compilateur n'essaiera alors pas, une fois les contraintes prioritaires respectées, d'avoir la fréquence la plus haute possible, ce qui permet de gagner un temps non négligeable lors de la génération du Bitstream en le divisant au minimum par deux (si tant

est que la fréquence choisie soit suffisante pour respecter les contraintes en termes de performances).

Conditions appropriées d'utilisation :

— Dans la grande majorité des cas, cette optimisation qui consiste à imposer sur certaines boucles un intervalle d'initialisation sous optimal, ne sera pas à mettre en place.

— S'il arrive que la fréquence globale d'un kernel diminue à cause d'une boucle dont le temps d'exécution rapporté est faible, il peut être judicieux d'en augmenter l'intervalle d'initialisation à l'aide de (IV.11) pour augmenter les performances totales de l'algorithme.

$$\#pragma\ ii\ valeur \tag{IV.11}$$

IV.2 Pipeline élémentaire

IV.2.1 Vectorisation

Le principe de la vectorisation est assez universel, et il est possible de l'effectuer sur la plupart des architectures actuelles. En effet, les CPUs, GPUs, FPGAs, et NPUs récents ont tous la possibilité de vectoriser certaines instructions.

En ce qui concerne les FPGAs, il est possible de tirer partie de ce concept pour augmenter le débit de calcul. Cette optimisation se retrouve en deux points clefs, que nous allons détailler ici.

IV.2.1.1 Vectorisation des work-items (NDRK, Intel)

La première possibilité, réservée uniquement aux NDRK, est de permettre à différents work-items de s'exécuter en SIMT (Single instruction Multiple Thread) avec une granularité fine.

Le compilateur OpenCL pourra alors traduire les opérations scalaires en opérations vectorielles, si tant est que la structure de l'algorithme le permette.

Il faut ajouter l'attribut (IV.12) en préfixe du kernel correspondant, où N correspond à la taille du vecteur désiré.

$$\_\_attribute\_\_((num\_simd\_work\_items(N))) \tag{IV.12}$$

Afin que l'outil accepte cette option de compilation, il faut par contre que le nombre de work-items soit divisible par le facteur vectoriel précédent, auquel cas la chaîne de compilation sera interrompue. Comme la compilation d'un fichier .cl se fait indépendamment de la partie hôte, il faut alors préciser, également en préfixe du kernel, la taille voulue du workgroup, qui doit respecter la condition ci-dessus.

IV.2.1.2 Vectorisation des paramètres d'entrées

D'une manière similaire, il est possible de vectoriser les paramètres d'entrée d'un kernel.

L'algorithme 8 permet d'illustrer comment réécrire le code sur l'hôte et sur le FPGA pour permettre au compilateur d'inférer une vectorisation des calculs. Le principe de l'algorithme illustré est d'additionner deux vecteurs.

Algorithme 8 : Vectorisation via les paramètres d'entrées

```
1 #define SIZE_ARRAY 1024
  /* Partie Hôte                                                    */
2 clCreateBuffer( ..., SIZE_ARRAY * sizeof(float), host_in1, ... );
3 clCreateBuffer( ..., SIZE_ARRAY * sizeof(float), host_in2, ... );
4 clCreateBuffer( ..., SIZE_ARRAY * sizeof(float), host_out, ... );
  /* Copie sur FPGA et lancement du kernel vector16                 */
  /* ...                                                            */
  /* Partie FPGA                                                    */
5 __kernel void vector16( __global float16 * restrict in1, __global float16 * restrict
   in2, __global float16 * restrict out) {
6 for ( int i = 0 to SIZE_ARRAY/16 - 1 ) {
7 |   out[i] = in1[i] + in2[i];
  |   /* calcul vectorisé par groupe de 16 réels                    */
8 }
```

Du côté de l'hôte, il n'y a en réalité rien à changer par rapport à la version classique. Une fois les *buffers* de réels flottants créés, et ils sont transférés sur le FPGA.

Du côte du FPGA par contre, les pointeurs en variable globale sont ici déclarés comme des *float16* au lieu des *float* classiques.

Le compilateur comprend à ce moment que l'on veut vectoriser ces accès, et va de lui-même prévoir les ressources en adéquation.

Ainsi, la boucle à la ligne 6 aura 16 fois moins d'itérations, puisque chaque opération de la ligne 7 somme 16 valeurs en parallèle.

Cette granularité de parallélisation est la plus efficace, en cela qu'elle a un surcoût en ressources presque négligeable par rapport aux autres solutions.

De plus, si l'on considère la bande passante entre le contrôleur mémoire du FPGA et ses cœurs de calcul, la taille du bus est souvent de 512 bits. Or, un mot de 16 réels flottants simple précision (*float16*) occupe 16*32 = 512 bits, et accéder directement à un mot permet de maximiser la bande passante du matériel sous-jacent, en plus d'optimiser les calculs par la vectorisation.

IV.2.1.3 Conditions appropriées d'utilisation

Les deux optimisations présentées précédemment peuvent en réalité être écrites à la main, tout autant que pourrait l'être le déroulage d'une boucle, mais cette étape est fastidieuse. L'intérêt majeur est donc au niveau du temps de développement, puisqu'il

n'y a pas à réécrire le code ni à s'occuper de changer la distribution des work-items. Le second avantage est que ces modifications minimalistes permettent à tout moment de revenir simplement à une version sans vectorisation, si l'on doit par exemple exécuter l'application sur une architecture ne la prenant pas en charge.

IV.2.2 Réplication (NDRK)

Dans le cas où l'on a un NDRK, le nombre de work-items est forcément strictement plus grand que 1. Dans ce cas, il est possible de répliquer le pipeline élémentaire. Ainsi, on augmente le nombre de cœurs de calcul, et les work-items pourront y être répartis. L'avantage majoritaire est donc de pouvoir partager entre tous les work-items d'un même work-group des éléments en mémoire locale.

Pour de plus amples détails, se référer à la section II.6.

Conditions appropriées d'utilisation :

Cette optimisation est à effectuer en dernier recours, et on lui préférera le pipeline ou le déroulage de boucle. En effet, dans le cas d'une réplication complète du pipeline, obtenue par l'ajout en préfixe du kernel de la commande IV.13, le surcoût en ressources est plus important que pour les autres techniques, car non seulement les mémoires sont répliquées, mais de plus la synchronisation des work-items nécessite plus de ressources puisqu'elle s'en trouve complexifiée.

$$\_\_attribute\_\_((num\_compute\_units(N))) \tag{IV.13}$$

Dans le cas où le recours à cette optimisation ne peut être évité, il faut alors veiller à ce qu'elle ne pénalise pas les performances générales de l'algorithme. En effet, augmenter le nombre de coeurs de calcul ajoute des contraintes sur les objets mémoires partagés entre tous les work-items, et, si l'algorithme est déjà limité par la bande passante, il est possible que la réplication ait un effet contre-productif et entraîne une baisse des performances.

IV.2.3 Mémoires locales

Comprendre et maîtriser la mise en mémoire locale d'objets sur FPGA est très souvent l'étape la plus ardue, en cela qu'une mauvaise implémentation sera souvent plus pénalisante que de ne pas utiliser de mémoire locale.

Nous avons formalisé ici, en complément de nombreux autres mécanismes amplement détaillés dans les documentations respectives des constructeurs de FPGA, deux mécanismes qui permettent de tirer parti efficacement des mémoires locales.

IV.2.3.1 Partition et découpage des objets

La première notion est la partition d'un objet en banque mémoire locale. En effet, nous avons évoqué précédemment le pipeline et le déroulage de boucle. Dans le premier cas de figure, la mémoire n'est pas répliquée, contrairement au second cas de figure.

Pourtant, il existe une solution pour éviter parfois d'avoir recours à cette réplication coûteuse en ressources.

Prenons l'exemple de la fonction local_ref de l'algorithme 9, et posons comme hypothèses que :

— la boucle (ligne 3) est de grande taille (SIZE_ARRAY > 256) ce qui empêche le déroulage complet,

— l'analyse de l'algorithme oriente vers un déroulage de cette boucle,

— cette même boucle empêche la vectorisation.

Algorithme 9 : Modification de la politique d'accès (mémoire locale)

```
1  #define SIZE_ARRAY 2048
   /* Version initiale                                          */
2  __kernel void local_ref( __global float * restrict in) {
3  for ( int i = 0 to SIZE_ARRAY - 1 ) {
4  |  tmp = global[i] ;
   |  /* calculs utilisant tmp                                  */
5  }
   /* Version optimisée                                         */
6  #define rep_factor 16 __kernel void local_opti( __global float * restrict in) {
7  local float local[SIZE_ARRAY / rep_factor][rep_factor] ;
8  #pragma unroll
9  for ( int i = 0 to rep_factor - 1 ) {
10 |  for ( int j = 0 to SIZE_ARRAY / rep_factor - 1 ) {
11 |  |  local[i][j] = global[i * rep_factor + j] ;
12 #pragma unroll rep_factor
13 for ( int i = 0 to SIZE_ARRAY - 1 ) {
14 |  tmp = local[i % rep_factor][i / rep_factor] ;
   |  /* calculs utilisant tmp                                  */
15 }
```

Alors dans ce cas, il est possible d'avoir, d'une part, recours à la mémoire locale et, d'autre part, d'optimiser sa structure pour permettre une intégration optimale dans l'algorithme. D'après les hypothèses de départ, notre boucle ne peut pas être déroulée entièrement, et on note alors rep_{factor} le facteur de réplication de cette boucle.

En implémentant une mémoire locale classique, elle sera répliquée pour permettre à la boucle déroulée d'y accéder avec le moins d'accès concurrent possible. L'optimisation définie est de découper notre tableau $local$ en rep_{factor} banques mémoires différentes, ce qui permet ensuite à chaque partie de la boucle déroulée d'avoir accès à une partie de ce tableau.

Le fait d'avoir partitionné le tableau en sous-tableaux permet à chaque boucle déroulée élémentaire d'accéder à sa section du tableau sans conflit d'accès. Il n'y a alors plus besoin de répliquer le tableau complet en mémoire locale, et le design généré consomme moins de ressources, en plus d'optimiser la bande passante locale.

Il convient de préciser que cette optimisation, dans le cas d'un NDRK, peut également être implémentée, et le déroulage de boucle sera remplacé par le lancement en work-items à l'échelle locale.

IV.2.3.2 Registre à décalage (SWIK)

Cette optimisation, définie ci-après, ne peut être implémentée que sur les SWIKs. L'idée est, dans le cas d'une boucle déroulée ou en pipeline, de pouvoir facilement partager entre chacune de ses itérations les données d'un tableau. Particulièrement efficace pour les algorithmes qui reposent sur des traitements en flux de données, cette optimisation s'appuie sur l'implémentation d'un registre à décalage.

La Figure IV.6 illustre ce dernier et le chapitre IV présente certaines implémentations de ce type d'optimisation.

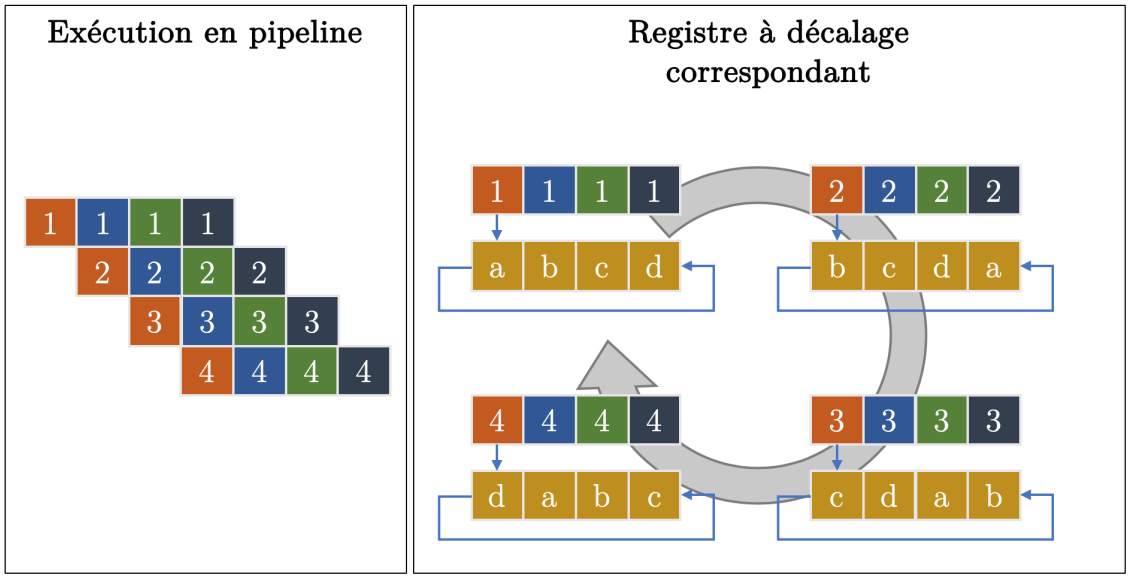


FIGURE IV.6 – Registre à décalage dans le cas d'une boucle en pipeline (les flèches représentent le flot des données)

Le principe est le suivant :

Chaque case d'un tableau est implémenté sur des registres différents, et un mécanisme garantit qu'à chaque cycle d'horloge, ou tout du moins à intervalle régulier, le contenu de ces registres et donc du tableau soit décalé d'une case, de manière circulaire.

Ainsi, l'objet mémoire est implémenté dans la zone mémoire la plus rapide et proche du cœur de calcul, et les ressources nécessaires à son utilisation sont minimales.

Dans le cas où l'algorithme se prête à ce genre d'optimisations, le gain global en termes de ressources et de performances est conséquent.

La Figure IV.6 illustre le cas où une boucle est en pipeline. Dans le cas où cette dernière serait déroulée, il est possible, en gardant ce registre à décalage, d'avoir plusieurs itérations déroulées qui accèdent en parallèle à une case différente du registre. Ainsi, l'ac-

cès est ici encore optimisé au maximum pour permettre aux données du tableau d'être accédées dans un flot continu sans concurrence d'accès.

IV.3 Mémoire globale et interface avec l'hôte

IV.3.1 Types de mémoires

Comme présenté à la section II.6, OpenCL propose différentes structures mémoires. En particulier, en ce qui concerne les mémoires globales, il en existe deux types : la mémoire $global$ et la mémoire $constant$, respectivement annotées du côté du kernel avec les mots clefs $\_\_global$ et $\_\_constant$.

Retenons que :

- $\_\_global$: il s'agit de la mémoire globale classique, donc de grande taille et avec une latence élevée. Il existe une mise en cache automatique tant chez Xilinx qu'Intel, similaire aux mécanismes de cache usuels sur CPU.
- $\_\_constant$: cette structure mémoire globale, réservée aux données en lecture seule, comprend une mise en cache des données de taille variable et paramétrable chargée dans la mémoire interne du FPGA.

Pour cette seconde structure mémoire, il est possible à la compilation de paramétrer la taille de ce cache interne (qui est par défaut de 16 KB), mais contrairement à la mémoire globale qui inclut divers mécanismes permettant de gérer les longues latences, un *cache miss* aura de lourdes conséquences au niveau de la performance sur une donnée $\_\_constant$. Dans le cas où le nombre de *cache miss* est conséquent, il vaudra mieux privilégier le stockage en $\_\_global\ const$ (voir ci-après) plutôt qu'en $\_\_constant$.

En plus de ces différentes zones mémoires, il existe des qualificatifs permettant d'aider le compilateur dans son choix d'optimisation, que nous allons présenter ci-après.

- $const$: permet de préciser que le pointeur correspondant pointe vers la mémoire globale, mais que seuls les ports en lecture sont utilisés à l'intérieur du kernel. Cela permet ainsi de réduire les ressources nécessaires à son implémentation, en n'intégrant pas les ports inutilisés en écriture [utilisation : $\_\_global\ const$].
- $volatile$: ce paramètre permet d'interdire la mise en cache automatique des données en mémoire globale. Cette option est à utiliser uniquement lorsqu'il n'y a pas de redondance de données dans le code. Encore une fois, cela permet de réduire la latence mais peut s'avérer pénalisant s'il y avait des redondances régulières.
- $restrict$: le compilateur OpenCL part du principe que chaque pointeur passé en argument peut pointer vers la même zone mémoire. Ajouter ce qualificatif à l'un d'eux, permet de préciser que celui-ci pointe vers une zone à laquelle il est le seul à pouvoir accéder. Cela permet de supprimer certains mécanismes garde-fous ajoutés par le compilateur en l'absence de ces précisions.

Aussi, il existe une certaine variétés d'options possibles pour les objets en mémoire globale, et le chapitre IV illustre les conséquences des différents choix liés aux spécificités de ces structures, tout en mesurant leur impact au niveau des performances.

IV.3.2 Partition et répartition des objets sur différentes banques mémoires

OpenCL laisse la liberté à chacun de ses constructeurs partenaires d'implémenter le matériel correspondant à chaque fonctionnalité, tant que la syntaxe est respectée. Du côté des constructeurs de FPGAs, la mémoire globale correspond souvent à de la DDR3, DDR4, ou encore à de la HBM2 pour certaines cartes haut de gamme, située à l'extérieur de la puce FPGA, et, suivant les configurations, chaque type de mémoire peut avoir plusieurs banques différentes.

Prenons comme exemple l'addition de deux vecteurs. En plaçant chacun d'entre eux sur une banque différente, nous pouvons y accéder en parallèle, sans conflit d'accès au niveau des bus mémoires.

Répartir différents objets mémoires sur des banques différentes, a donc comme conséquence d'augmenter la bande passante maximale entre les kernels et la mémoire globale du FPGA.

IV.3.3 Communication entre kernels (*pipes* et *channels*).

L'un des avantages majeur du FPGA est la présence de blocs RAM dans sa fabrique interne. Il existe ainsi des solutions qui permettent d'implémenter des mécanismes de communication efficace entre deux fonctions OpenCL qui s'exécutent sur le FPGA.

Le principe est simple. En implémentant des files premier entré premier sorti (*First In First Out*) (FIFO), dont la profondeur et la largeur d'une donnée élémentaire peuvent être paramétrées, il est possible d'éviter, comme illustré à la Figure IV.7, des allers-retours entre kernels et mémoire globale, qui encombreraient inévitablement les communications sur ces bus mémoires.

Ces files FIFO sont implémentées sur les blocs RAM internes, ce qui permet de tirer partie de la faible latence de ce type de mémoire.

Ainsi, il convient d'analyser au préalable les différents chemins de données pendant une exécution type d'un algorithme, afin d'implémenter, si nécessaire, ces mécanismes efficaces s'appuyant sur le mécanisme du producteur/consommateur, où une fonction produit des ressources qui sont ensuite consommées par une autre fonction.

IV.4 Caractérisation des leviers d'optimisation

Le standard OpenCL propose un très grand nombre d'optimisations possibles, et leur utilisation pour cibler une architecture FPGA peut s'avérer complexe. C'est pourquoi nous avons pu voir dans cette partie les conditions appropriées d'utilisation, qui permettent d'expliciter dans quels cas de figure ces optimisations peuvent s'avérer pertinentes.

La mise en place de la méthodologie à la section V.2 s'appuie sur ces caractérisations, et la Figure V.4 quantifie pour certaines optimisations leur influence sur la consommation en ressources ou sur leur efficacité.

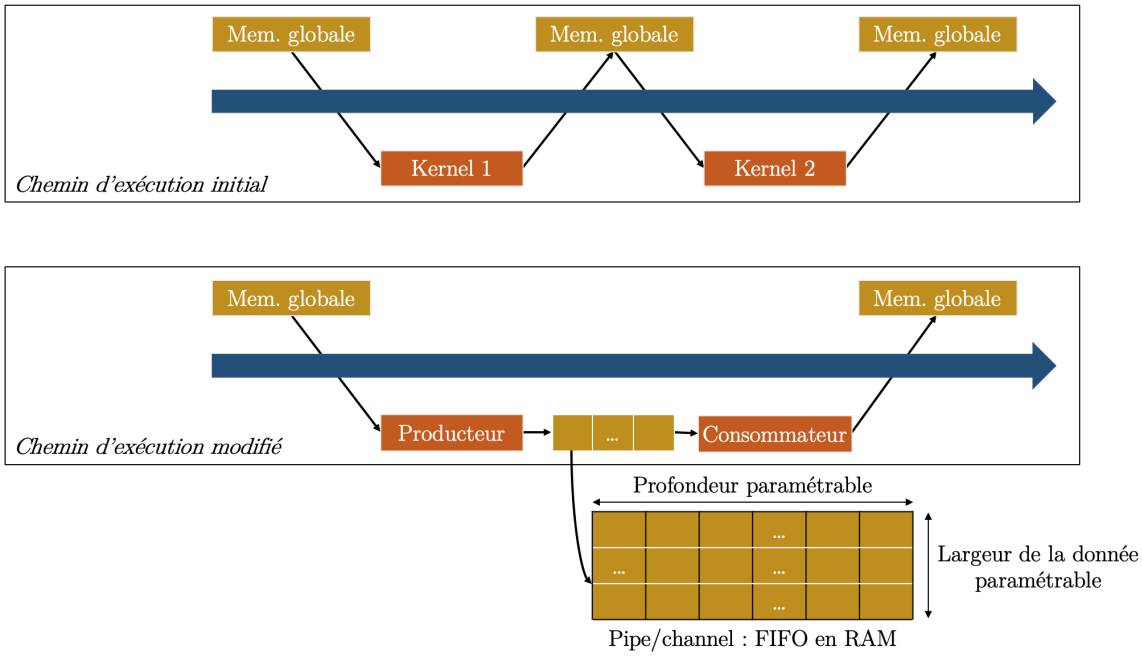


FIGURE IV.7 – Utilisation des pipes/channels pour une communication efficace entre kernels.

Chapitre V

Exploration du champ des optimisations

Sommaire

Nous finalisons notre méthodologie d'accélération d'algorithmes en OpenCL sur FPGA dans ce chapitre.

La base du modèle ayant été posé dans les chapitres précédents, à savoir les définitions des métriques et des optimisations élémentaires, nous allons expliquer ici notre principe d'exploration itérative des optimisations.

Les leviers d'optimisation définis au IV ayant été testées sur des cas d'école, nous devons établir un contexte de validation afin de pouvoir les évaluer, dans la Partie 3 sur un panel d'applications réelles.

Nous avons choisi de nous placer dans un contexte de Pareto, notion que nous introduirons en première partie du chapitre.

Notre méthodologie s'insère dans un cadre d'optimisation multi-critères. Nous nous plaçons en effet dans le cas où nous avons des contraintes au niveau des performances brutes (temps d'exécution) ainsi qu'au niveau des ressources consommées. Suivant le cahier des charges d'une application donnée, il est alors possible de définir un domaine des solutions acceptables, et l'objectif de notre méthodologie est de quantifier les différents leviers d'optimisations qui permettent, suivant les contraintes, de se rapprocher le plus rapidement possible de ce domaine.

La deuxième partie de ce chapitre sera consacrée à expliquer :

— comment explorer les optimisations possibles avec notre méthodologie AAA d'accélération d'algorithmes,

— le fonctionnement de notre Noyau d'Optimisation.

Nous conclurons ce chapitre par la présentation synthétique du Manuel d'utilisation de notre méthodologie AAA.

V.1 Solutions de Pareto : optimalité locale sous contraintes

V.1.1 Introduction des notions utiles

Dans ses études sur l'efficacité économique et la distribution des salaires, Vilfredo Pareto (1848 - 1923) énonce le concept désormais connu sous le nom d'**efficacité** ou **optimum de Pareto**, qui correspond à l'état où, dans un groupe d'individu, il n'est plus possible d'améliorer la situation de l'un d'entre eux sans dégrader celle d'au moins un autre [Pareto, Vilfredo, 1909].

L'énoncé mathématique pourrait se traduire ainsi : considérons une économie avec *n* états possibles $E = e_1, \ldots, e_n$ et un ensemble de *m* acteurs $A = a_1, \ldots, a_m$. Chacun de ces acteurs a une fonction d'utilité, qui lui permet d'avoir à titre individuel une relation de préférence $>_{a_i}$ sur E.

Ainsi, un état dit pareto-optimal est un état $e \in E$ tel qu'il n'existe pas d'autre $e' \in E$ tel que :

$$\begin{cases} \forall a \in A \ , e' \geqslant_a e \\ et \\ \exists a \in A \ / e' >_a e \end{cases}$$

Concrètement, un état qui est pareto-optimal ne peut donc être amélioré pour tous les acteurs du groupe, car cela entraînerait une baisse de la préférence d'au moins un acteur du groupe considéré.

Pour un système donné, l'ensemble des états pareto-optimaux forme ce que l'on appelle la **frontière de Pareto**.

Si, à partir d'un état, on peut trouver un état qui améliore au moins un acteur du groupe, et n'en n'aggrave aucun autre, alors on peut dire que ce second état **domine** au sens de Pareto le premier. Par contre, il n'est pas possible de parler de domination de Pareto entre deux états pareto-optimaux.

V.1.2 Application à la démarche d'optimisation

Ce principe, à l'origine pensé pour le domaine de l'économie, est couramment utilisé pour l'analyse des compromis dans une démarche d'optimisation multi-critères [Hendriks et al., 2011], [Jakob and Blume, 2014], [Emmerich and Deutz, 2018].

Nous nous sommes ainsi proposés d'adapter à notre démarche d'optimisation OpenCL sur FPGA les différentes notions liées aux solutions de Pareto, en dressant les correspondances suivantes :

— l'ensemble des acteurs devient l'ensemble des paramètres possibles pour optimiser une application ,

— la relation de préférence sur E devient la minimisation des ressources utilisées et du temps d'exécution,

— les états possibles de l'économie deviennent les optimisations réalisables.

Ainsi, pour un algorithme donné sur un FPGA précis, la Figure V.1 permet d'illustrer l'ensemble des optimisations réalisables.

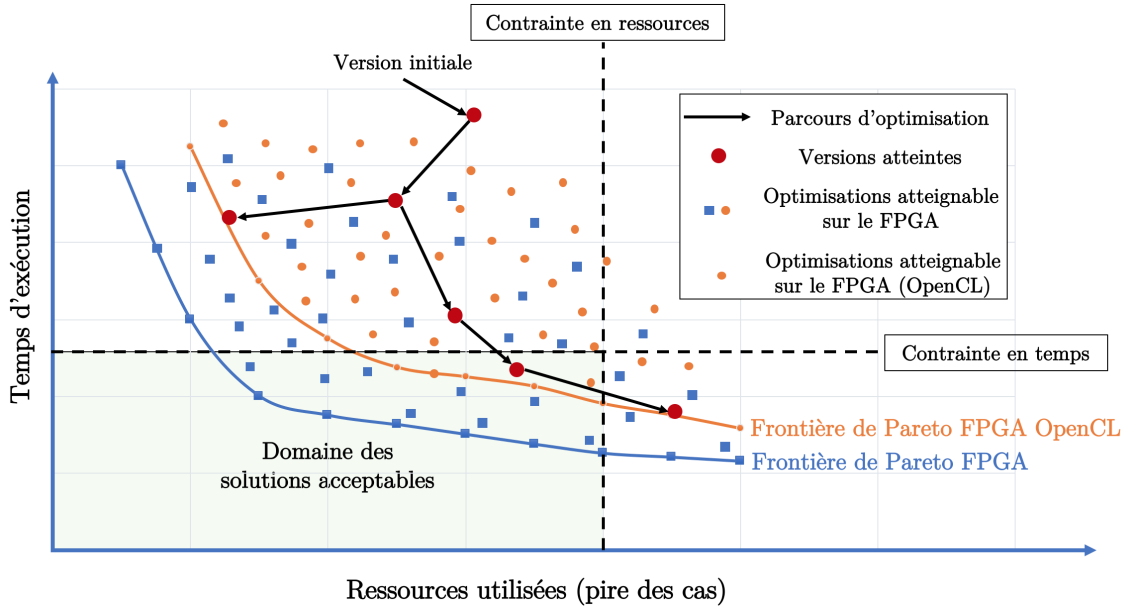


FIGURE V.1 – Illustration des frontières de Pareto sur un FPGA générique.

La frontière de Pareto la plus optimale correspond à celle qui n'est atteignable qu'avec les outils les plus bas niveaux, et l'utilisation des différentes approches haut niveau res-

treint les paramètres sur lesquels on peut jouer, et réduit ainsi l'ensemble des optimisations atteignables. La conséquence directe est une frontière de Pareto "réduite" qui englobe moins d'optimisations.

Ainsi, les carrés et les points sur cette figure représentent l'ensemble des optimisations atteignables sur la technologie cible, quelque soit l'outil utilisé, alors qu'avec l'utilisation d'OpenCL, seules les points sont accessibles.

Cette restriction de l'ensemble des possibles est l'une des contreparties de la facilité gagnée par l'utilisation de ces approches.

Il est donc théoriquement possible d'atteindre tous les points situés au niveau de la frontière de Pareto correspondante, et les résultats présentés au chapitre VI mettront en valeur les différents parcours d'optimisations choisis sur des graphes de Pareto.

V.1.3 Caractérisation d'une "bonne" optimisation

Les axes des différentes figures de Pareto que nous utilisons pour illustrer nos résultats sont d'une part le temps d'exécution, d'autre part les ressources utilisées par l'implémentation sur la cible choisie.

Le premier critère est donc celui de la performance brute, alors que le second peut être plus ou moins important selon les cas d'applications. Afin de garder une cohérence avec le modèle *roofline* présenté à la section III.2.4.3, la notion de ressources utilisées correspond à l'agrégation, suivant le pire des cas, des différents blocs élémentaires accessibles à l'utilisateur d'un FPGA.

Suivant la finalité d'un algorithme, la notion de "bonne" optimisation peut alors varier. Certaines applications ont une contrainte forte en ressources, car une partie du FPGA doit par exemple pouvoir être utilisée pour d'autres traitements, alors que, dans d'autres cas d'utilisation, le temps d'exécution est prioritaire, et il n'y a pas de limitation particulière en ressources. On peut observer sur la Figure V.1 le domaine des solutions acceptables en fonction de deux contraintes arbitraires en ressources et en temps, et les marques à l'intérieur de cette zone représentent l'ensemble des solutions acceptables.

La méthodologie d'optimisation d'algorithmes dépend donc fortement de l'existence de ces contraintes. En effet, s'il n'existe qu'une contrainte en ressources, alors les solutions à privilégier sont celles proches de la frontière de Pareto correspondante, mais dans la partie gauche du graphique, alors que s'il existe une contrainte en temps, il s'agira de privilégier la partie la plus basse donc à droite sur notre figure.

Dans le cas où il y a nécessité d'imposer deux contraintes, l'ensemble des solutions acceptables est donc restreint à un rectangle, que l'on retrouve également dans la partie inférieure gauche du graphe de Pareto.

Il convient donc de bien cerner quelles sont les contraintes prioritaires avant d'entamer une démarche d'optimisation, car de ceux-ci dépend cette notion de "bonne" optimisation.

V.1.4 Exploration des optimisations et sous-optimalité temporaire

L'exploration des optimisations, qui consiste à suivre une méthodologie, comme celle présentée à la section V.2, est illustrée aux chapitres VI, VII, et VIII. En partant d'une version initiale, l'objectif est de se rapprocher et, si possible, d'atteindre les solutions les plus pareto-optimales.

En définissant une méthodologie d'accélération d'algorithme en OpenCL, nous pourrions être tenté de raisonner uniquement en s'intéressant à la relation de domination au sens de Pareto.

Il peut arriver qu'au cours de l'exploration des solutions, on se retrouve à un moment donné avec une divergence des branches d'optimisations (voir Figure V.2), dont forcément plusieurs (Branches 1 et 3) permettent d'atteindre à l'étape suivante immédiate des optimisations moins paréto-optimales que les autres (Branche 2), alors qu'elles deviendront plus tard plus optimales.

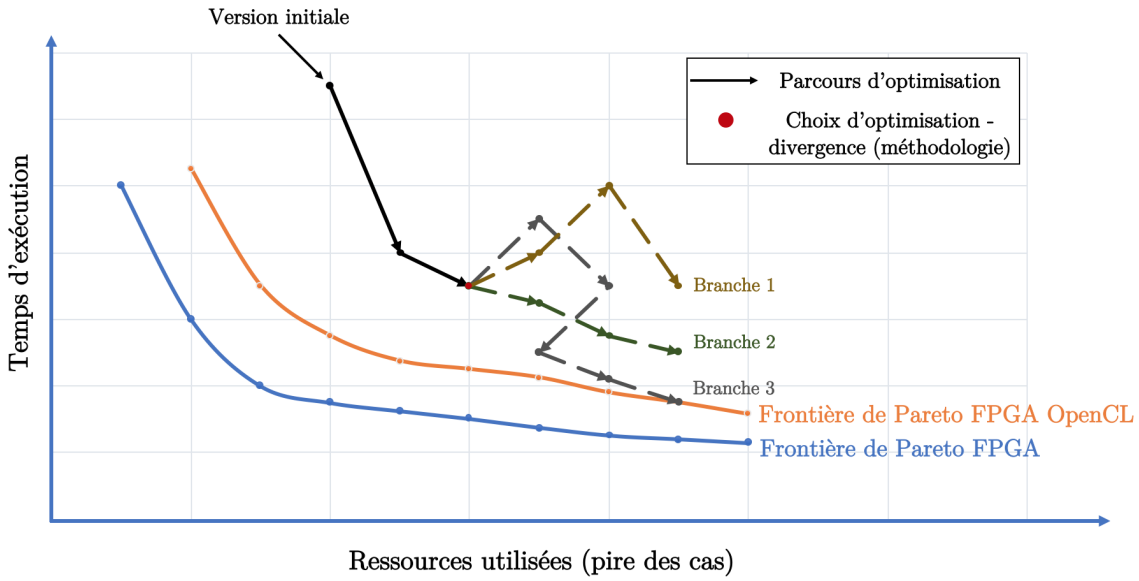


FIGURE V.2 – Exploration des optimisations possibles et divergence des branches.

Il faut donc veiller à ne pas délaisser trop hâtivement une branche d'optimisation même si elle apparaît temporairement comme sous-optimale comme la branche 3 dans la Figure V.2. La difficulté est alors d'arriver à prévoir si une branche sous-optimale peut potentiellement redevenir optimale au fur et à mesure de notre parcours d'exploration.

V.1.5 Limites du critère et conséquences pour la méthodologie

Limite 1 :

Comme pour la définition du modèle *Roofline*, l'approximation par le pire des cas du pourcentage de ressources utilisées peut induire une forte non-linéarité au niveau de la stratégie d'exploration, comme illustré par la branche 3 dans la Figure V.2. En effet, il existe de nombreux cas de figure dont certains sont présentés au IV, où la partition d'un objet en mémoire locale peut, de concert avec un déroulage de boucle cohérent, améliorer fortement les performances d'une application. Pourtant, si on ne s'occupe que de la mise en cache en mémoire locale de l'objet correspondant, le surcoût lié à cette optimisation la rend moins pareto-optimale que l'optimisation sans mise en cache, et on pourrait alors croire que cette solution ne doit pas être envisagée.

Solution 1 :

Il faut donc, pendant l'ébauche de la méthodologie, identifier au mieux les différentes optimisations qui permettent, groupées, d'obtenir un meilleur temps d'exécution, tout en utilisant moins de ressources. Cela permettra d'une part d'éviter des optimisations pouvant être jugées comme peu intéressantes car peu efficaces seules, et d'autre part de nuancer les plus gros effets de non-linéarité, pouvant fausser nos décisions durant la stratégie d'exploration.

Limite 2 :

Au moment d'une divergence de branches, si on définit comme stratégie d'exploration celle qui s'attache à privilégier la branche la plus paréto-optimale, il sera alors possible que des branches temporairement sous-optimales soient délaissées alors qu'elles auraient été dominantes au sens de Pareto par rapport aux autres branches plus tard dans l'exploration.

Solution 2 :

Il est donc clair qu'il ne faut pas raisonner immédiatement après une divergence de branche, pour réduire les cas où il faut un petit nombre de paramètres pour pleinement tirer partie d'une optimisation en apparence sous-optimale. Une des solutions correspond à la solution 1 présentée ci-dessus. En groupant certains paramètres d'optimisations, on diminue les cas de figures où une branche paraît temporairement sous-optimale.

L'autre solution prise en compte ici est d'incorporer des mécanismes de rebouclages permettant de rattraper certains paramètres d'optimisations délaissés plus tôt dans la stratégie d'exploration car peu efficace à ce moment.

Limite 3 :

Dans le cas de contraintes fortes, il est possible que le domaine des solutions acceptables n'intersecte pas l'ensemble des optimisations atteignables sur le FPGA avec OpenCL. Cette limitation, propre à l'utilisation d'outils à haut niveau d'abstraction, peut cependant être contournée si nécessaire.

Solution 3 :

Une des solutions est de passer outre les outils haut niveau. Sans pour autant s'en détacher complètement, il est possible avec les outils Intel et Xilinx d'intégrer, dans un design OpenCL, une fonction développée en Langage de Transferts de Registres (*Register Transfer Level*) (RTL), donc optimisée à souhait.

Dans le cas où cela n'est pas possible uniquement en OpenCL, cette approche hybride permet de satisfaire les contraintes imposées.

V.2 Mise en forme de la méthodologie d'accélération d'algorithmes en OpenCL sur FPGA

V.2.1 Description de la stratégie générale

Dans nos travaux, notre principale priorité est de minimiser le temps d'exécution, et la minimisation des ressources utilisées est donc moins prioritaire. Au niveau des frontières

de Pareto, on cherche donc à être le plus bas possible, quitte à dériver sur la partie droite du graphique.

Plus généralement, nous présentons dans cette section le processus itératif général d'optimisation, qui décrit les différents niveaux de validation des implémentations, puis nous illustrons le choix des différentes optimisations détaillées au Chapitre IV, ce qui constitue notre méthodologie d'accélération d'algorithme sur FPGA via OpenCL.

V.2.2 Processus itératif général

A partir d'un algorithme de référence, nous définissons un graphe d'état qui souligne les étapes nécessaires à son optimisation.

Illustré à la Figure V.3, il y a trois types de représentation dans un flot d'optimisation classique d'un code OpenCL sur FPGA.

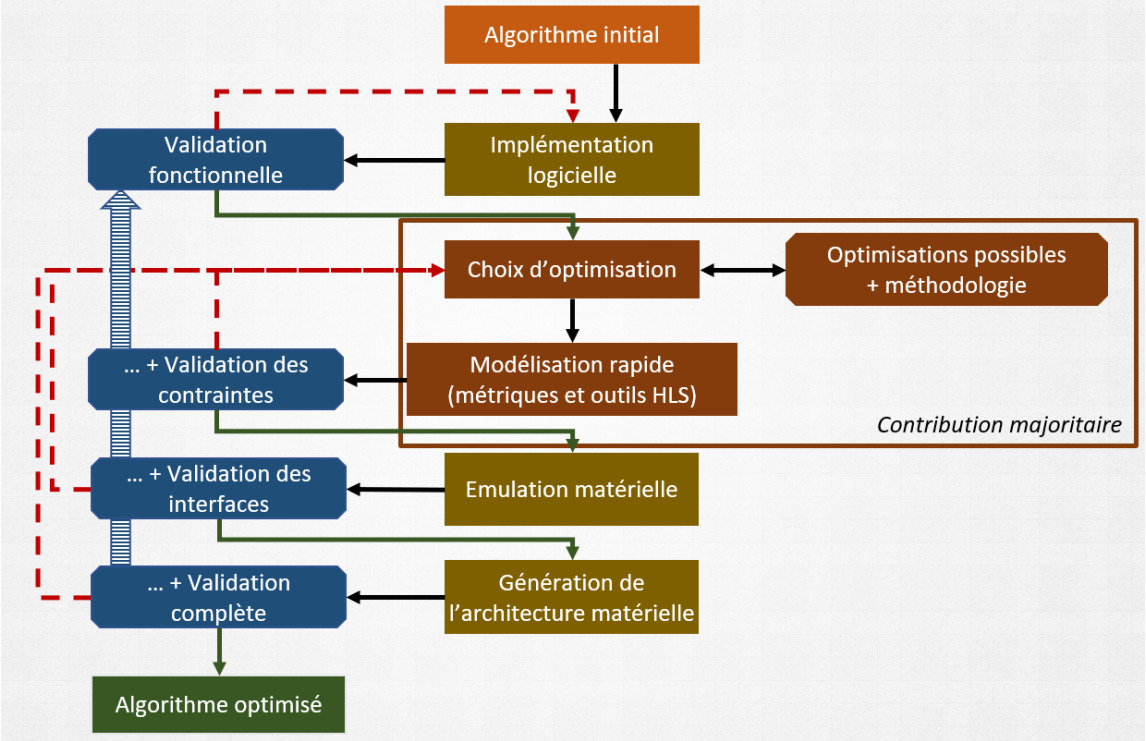


FIGURE V.3 – Principe de l'exploration itérative des optimisations.

On retrouve donc :

— l'implémentation logicielle : cela correspond à traduire en OpenCL l'algorithme de référence, avec la syntaxe correspondante, sans utiliser d'optimisation, et à exécuter ce programme sur l'hôte uniquement. Cette étape permet de vérifier la fonctionnalité de notre code OpenCL initial.

— l'émulation matérielle : une fois la fonctionnalité du code validée, il est possible d'émuler le matériel correspondant à l'aide des outils Xilinx et Intel. Cette étape, dont la compilation reste plus longue (5 min) qu'une compilation software, est tout

de même rapide par rapport à une génération complète du Bitstream du FPGA (de 1 à plus de 10h). Cette étape permet d'avoir une idée préliminaire de certains goulots d'étranglement qui peuvent apparaître une fois le design réel généré, mais il faut garder à l'esprit que le taux d'expansion de cette étape peut rapidement être important, vu que le comportement du FPGA sera simulé à chaque cycle d'horloge par le CPU.

— la génération du matériel : étape la plus longue dans le flot de conception d'un FPGA, ce n'est qu'une fois cette étape terminée que nous pouvons tester notre optimisation et en analyser l'efficacité réelle.

Avec les différentes métriques présentées à la section III.2, nous avons ajouté une étape intermédiaire (*Modélisation rapide*), qui, à partir de l'analyse simple du code, permet de modéliser rapidement les kernels sur le FPGA. Cela permet de créer un nouveau flot de rétroaction manuel pour caractériser la pertinence des optimisations choisies, qui est plus rapide que leur émulation matérielle.

Les trois étapes, entourées par l'encadré *Contribution majoritaire*, regroupent la majorité de notre analyse méthodologique, et elles sont détaillées à la sous-section suivante (V.2.3).

Aussi, notre méthodologie, qui se greffe sur le flot usuel de conception d'une application sur FPGA, consiste, après chaque nouvelle optimisation implémentée, à vérifier la validité de cette dernière, et le gain, tant en ressources qu'en performances.

Toutefois, pour des cas d'utilisation simples ou déjà identifié, il est possible d'implémenter un certain nombre d'optimisations en une seule fois, mais cette démarche nécessite d'avoir suffisamment pu expérimenter avec les différents outils pour garantir qu'il ne subsiste pas une zone non optimisée qui rendrait sous-optimal le regroupement des optimisations choisies.

V.2.3 Noyau d'Optimisation (*contribution majoritaire*)

Cette sous-section présente et illustre, à la Figure V.4, le choix possible des optimisations sur FPGA en utilisant OpenCL.

La lecture de ce graphe d'états se fait ainsi :

— De haut en bas, les étapes clefs d'optimisation sont dans la partie gauche du graphique.

— Pour chacune de ses étapes, on retrouve sur la droite un certain nombre de sous-catégories et d'optimisations, qui sont pour la plupart annotées par des indicateurs de ressources et de performances.

— *Indicateurs* : pour les performances, une bonne optimisation, représenté par un $+$ vert entouré d'un cercle, diminue le temps d'exécution. Pour les ressources, un $-$ vert entouré d'un carré signifie que cette optimisation consomme moins de ressources. L'absence d'indicateur signifie que la performance d'optimisation dépend de la pertinence de celle-ci suivant les cas de figures.

— Nous avons déjà évoqué qu'il existait plusieurs types de kernels en OpenCL : les SWIKs et les NDRKs. Aussi, comme détaillé dans la légende de la Figure V.4, la forme d'une optimisation correspond à sa compatibilité suivant le type de kernel.

Les concepts énoncés sont ceux déjà abordés et détaillés au Chapitre IV.

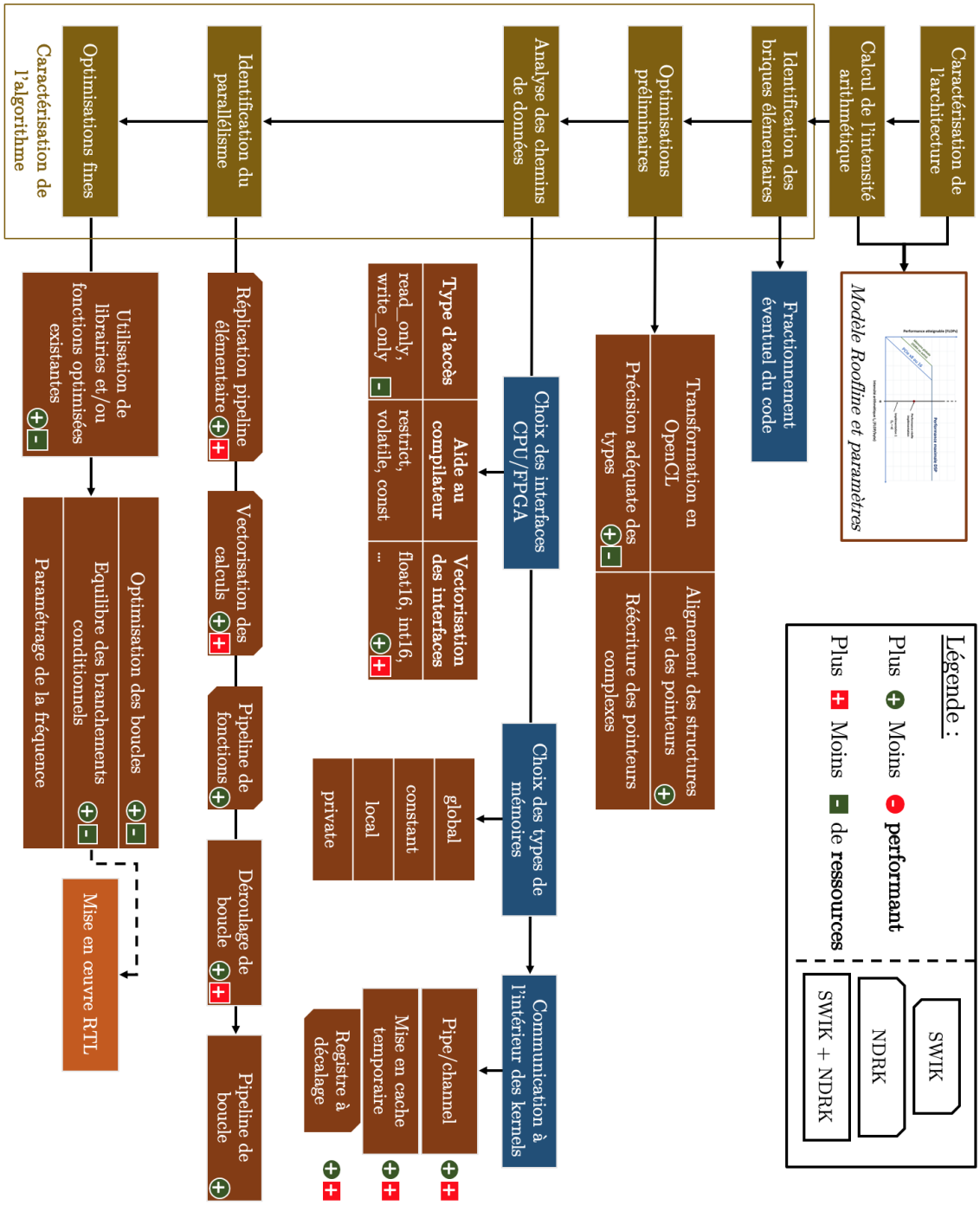


FIGURE V.4 – Optimisations réalisables avec indices de performance et de consommation en ressources : Noyau d'optimisation

La méthodologie que nous avons mise en place, ne donne pas de priorité forte à une

quelconque optimisation, car nous laissons la liberté au développeur de décider, en s'appuyant sur les concepts d'optimisations précédemment énoncés, de la marche à suivre en fonction de l'évolution des performances des optimisations implémentées.

Nous recommandons cependant :

— si le goulot d'étranglement se fait au niveau de l'accès aux données, de s'intéresser aux différents types de mémoires, afin d'en choisir le plus pertinent suivant la politique d'accès.

— si les calculs sont limitants, et dans le cas où les possibilités d'OpenCL ont toutes été déjà exploitées, d'envisager d'implémenter en RTL la fonction la plus critique. Réduire l'abstraction du langage peut alors offrir plus de leviers d'optimisations et donc potentiellement améliorer les performances.

De même, si les calculs sont limitants, et dans le cas où les possibilités d'OpenCL ont toutes été déjà exploitées, il faut envisager d'implémenter en RTL la fonction la plus critique, afin, en réduisant l'abstraction de notre langage, de gagner des leviers d'optimisations et donc d'améliorer les performances.

V.2.3.1 Choix de la zone mémoire adéquate

Il existe de nombreux types de mémoires sur FPGA, vues au travers du prisme OpenCL. Nous avons choisi de synthétiser ici les différentes caractéristiques de celles-ci, présentées plus en détail aux sections IV.3 et IV.2.3.

— $\_\_constant$: l'outil implémente une structure de cache optimisée, très pénalisée par les défauts de cache. A utiliser quand les accès sont parfaitement réguliers.

— $\_\_global$: permet également un mécanisme de cache mais cette fois ci avec peu de pénalité en cas de défauts de cache. Ce fonctionnement est similaire à la mise en cache sur CPU. A utiliser quand les accès ne sont pas complètement réguliers, mais suffisamment quand même.

— $\_\_global\ volatile$: permet de désactiver le cache précédent. Convient aux accès très majoritairement aléatoires (très grande probabilité de défaut de cache). Cela permet surtout d'économiser des ressources matérielles, bien qu'il existe également un léger gain en latence.

— $\_\_local$: implémenté sur les blocs RAM du FPGA, cette zone convient bien aux accès aléatoires à latence faible. La difficulté est de pré-charger correctement les données dans cette zone mémoire limitée.

— $\_\_private$: similaire à la mémoire locale.

Il est donc crucial, pour comprendre les perspectives d'optimisation concernant la mémoire, de connaître le motif d'accès aux données dans l'algorithme (accès aléatoire, régulier, pseudo-périodique, ...) .

Il peut toutefois arriver que, bien qu'on ait choisi la structure mémoire la plus optimale, et que le parcours d'optimisation soit terminé, que l'on n'arrive pas à obtenir de performance suffisantes.

Si la limitation vient de la mémoire, il peut alors être intéressant de changer la technologie de cette dernière, et d'analyser la pertinence d'autres technologies de mémoire

comme la HBM2 ou la QDR en fonctions de leurs caractéristiques (bus de lecture/écriture parallèles, débits, efficacité des accès aléatoires, ...).

V.2.3.2 Types de parallélisme

Au moment du choix du parallélisme, le côté hybride de la programmation OpenCL entrouvre de multiples possibilités puisqu'il est possible de faire coïncider parallélisme de données, de tâches, et de fonctions, à l'intérieur d'une même application.

Le choix des optimisations doit se faire en fonction des ressources disponibles, de l'indépendance des données, mais également en fonction de la granularité voulue du parallélisme, de la plus forte (Réplication du pipeline élémentaire) à la plus fine (Pipeline de boucle).

En effet, la réplication d'un pipeline élémentaire impose également une duplication des mémoires et des ports correspondants, ce qui certes améliore les performances mais impose un surcoût matériel important, alors que la mise en pipeline d'une boucle est la solution la plus efficace pour mettre en commun un même matériel généré.

Choisir le type d'expression du parallélisme de données dépend souvent de la contrainte en ressources.

V.3 Manuel d'utilisation de notre méthodologie d'accélération

Notre méthodologie d'accélération d'algorithmes en OpenCL sur FPGA consiste à produire à partir d'un algorithme initial, une ou plusieurs versions optimisées, afin, après les étapes de génération matérielle et de validation, de les exécuter sur un FPGA.

Notre objectif est évidemment de sélectionner la ou les meilleures optimisations, eu égard aux critères pertinents dans le contexte de l'accélération, que sont le temps d'exécution et les ressources utilisées. Or la génération matérielle d'un Bitstream, prend en moyenne 6h30 pour des algorithmes complexes, ce qui est sans comparaison avec la durée de compilation d'un algorithme sur CPU.

Néanmoins comme nous l'avons souligné, notre exploration s'effectue de façon itérative : nous réitérerons le processus jusqu'à trouver la meilleure combinaison d'optimisations possible, ce qui prendra un temps très conséquent en terme de temps de compilation. L'intérêt d'une modélisation rapide des performances estimées est d'éviter au maximum de passer par l'étape de génération de Bitstream pour chaque levier d'optimisation.

L'étape la plus importante qui a constitué notre contribution majoritaire est l'étape de choix et d'application des modélisations, ou **Noyau d'Optimisation**. Nous soulignons que parmi les optimisations possibles, certaines sont utiles quelque soit l'algorithme, elles seront donc réalisées systématiquement. Il s'agit par exemple de l'optimisation de l'adéquation de la représentation des nombres. Ce type d'optimisation ne fera pas évidemment l'objet d'un processus de sélection.

Manuel d'utilisation de notre méthodologie AAA :

Étape 1 : **Modélisation** de l'architecture cible (Modèle *Roofline* FPGA).

Étape 2 : **Identification** des fonctionnalités à adapter.

Étape 3 : **Implémentation** d'une première version fonctionnelle.

Étape 4 : **Modélisation rapide** à l'aide des métriques et des outils HLS.

Étape 5 : **Choix et implémentation** des optimisations pertinentes.

Étape 6 : **Rebouclage** suivant leurs performances.

Étape 7 : **Test matériel** et mesure des performances.

Nous détaillons dans la partie suivante la pertinence de certains choix d'optimisations sur des applications réelles.

Troisième partie

Application et évaluation de la méthodologie

Remarques introductives

Cette troisième partie est consacrée à l'application de notre méthodologie d'accélération sur un large panel d'algorithmes, dont certains sont inclus dans des projets d'envergure en environnement réel. Les thématiques abordées sont la reconstruction tomographique, la modélisation d'environnements radar et de système d'écoutes électromagnétiques, et des algorithmes adaptés de benchmarks d'optimisation. Dans cette section introductive, nous présentons les notations, les différentes cartes, plateformes, et outils utilisés pour la mise en œuvre de nos implémentations, et y définissons également notre protocole de test.

Dans le chapitre VI consacré à la présentation d'un algorithme de rétroprojection dans le domaine de la reconstruction tomographique, nous comparons les performances de nos algorithmes optimisés avec les outils OpenCL d'Intel, aux performances d'algorithmes sur GPUs, considérés comme architecture de référence du domaine, ainsi qu'à une version optimisée sur VHDL d'un algorithme similaire. Le Chapitre VII permet de détailler le parcours d'optimisation suivi pour l'accélération d'algorithmes industriels développés à Thales DMS France, et les performances obtenues sur un FPGA Xilinx y sont évaluées, notamment par le prisme de la facilité de développement d'un code OpenCL par rapport à un algorithme de référence en C. Le chapitre VIII permet de présenter les résultats d'une démarche d'optimisation éclairée de code OpenCL sur FPGA, appliquée à un panel représentatifs d'algorithmes, issu de suites de benchmark optimisées pour les CPUs et les GPUs. Ces différentes implémentations ont pour objectif de souligner, suivant les contraintes, les optimisations les plus pertinentes.

Brève chronologie de notre expérimentation

Les premiers travaux et évaluations ont été menés sur la DE1-SoC, dotée d'un SoC comprenant un FPGA Intel Cyclone V SE couplé à un CPU ARM Dual Core. Ce FPGA d'entrée de gamme, utilisé avec la version 16.0 de la suite Quartus d'Intel nous a permis d'obtenir un certain nombre de résultats, notamment dans la caractérisation des latences des différentes structures mémoires vues par le modèle de programmation OpenCL [Martelli et al., 2019b]. Sur cette version de l'outil, il était impossible de récupérer ces informations depuis la chaîne de compilation OpenCL. Depuis, certains travaux ont été repris sur un FPGA Arria10 d'Intel, avec le SDK d'Intel pour OpenCL, version 18.0. Entre ces deux versions, les outils ont significativement évolué. Certaines optimisations qui devaient êtres implémentées manuellement, comme par exemple, la fusion des boucles imbriquées (voir section IV.1.2.4) peuvent désormais être automatiquement implémentées à l'aide d'une

directive de pré-compilation OpenCL, ce qui évite de devoir substantiellement réécrire le code en OpenCL.

Intel et Xilinx proposent régulièrement des mises à jours majeures de leurs outils OpenCL. Toutefois, les drivers distribués par les fournisseurs des cartes FPGAs ne suivent pas forcément la même cadence. Par exemple, en ce qui concerne notre FPGA Arria 10, le fournisseur Rexel ne propose pour le moment pas de compatibilité avec la version 19 de Quartus.

C'est pourquoi nous avons décidé de rester sur une version stable des outils, dont le détail des versions sera détaillé ci-après.

Architectures de calculs utilisées

Si nos principaux travaux se sont focalisés sur les FPGAs, nous avons néanmoins implémenté ou utilisé des versions optimisées de nos algorithmes sur différents CPUs et GPUs pour effectuer des comparaisons exhaustives de ces architectures. Ces dernières sont présentées dans le tableau récapitulatif ci-dessous (Tableau V.1), auquel le lecteur pourra se référer tout au long des chapitres qui suivent.

TABLE V.1 – Architectures utilisées et caractéristiques

	CPU	**GPU 1**	**GPU 2**	**FPGA1**	**FPGA2**	**FPGA3**
Fabriquant	Intel	Nvidia	Nvidia	Intel	Intel	Xilinx
Famille	Xeon	GTX	Jetson	Cyclone V	Arria10	Kintex Ultrascale
Modèle	E5-2667	1080Ti	TX2	DE1-SoC	10 GX 1150	KCU115
Année	Q2 2012	Q1 2017	Q1 2017	Q2 2014	Q4 2013	Q4 2013
Finesse de gravure (nm)	22	16	16	28	20	20
Prix (€)	1552	850	600	250	4700	3500
Fréquence (GHz)	2.9	1.48	0.3	0.3	0.3	0.3
Cœurs physiques	6	3584	256	–	–	–
Mémoires globales (GB)	96	11	8	1	8	16
Type de mémoire	DDR3	GDDR5X	LPDDR4	DDR3	DDR4	DDR4
Bande passante correspondante (Gb/s)	1.6*4	484.4	58.4	1.6	2.4	2.4*4
LUT	–	–	–	36'291	427'200	663'360
Pins I/O	–	–	–	–	826	832
Blocs RAM	–	–	–	397	2713	6480
Bits mémoire	–	–	–	4'065'280	55'562'248	18'360'000
DSP	–	–	–	87	1518	5520
Performance FP32 (TFLOPS)	0.5	10.61	0.665	0.067	1.366	2.682

Ce tableau met en regard certaines caractéristiques de ces différentes architectures, mais il convient ici d'en nuancer certaines.

Par exemple, la performance FP32 sur FPGA n'est calculée qu'à partir du nombre de DSPs, avec une fréquence d'utilisation théorique. En réalité, utiliser toutes les ressources d'un FPGA impose une contrainte forte sur la fréquence, qui pourra donc fortement diminuer par rapport à la valeur d'utilisation type, réduisant les performances annoncées. Aussi, la difficulté de caractériser les ressources logiques d'un FPGA (hors DSP) rend la définition d'une performance réelle assez complexe, comme nous l'avions vu à la section III.2.4.3. Cet indicateur n'est donc pas suffisant pour l'évaluation des performances d'une architecture.

D'une architecture ou famille de carte à l'autre, le type de technologie mémoire ou la taille des bus par exemple peuvent avoir des répercussions significatives sur les performances. Ici encore, il faut considérer les valeurs présentées au Tableau V.1 comme des indicateurs potentiels des performances d'une architecture donnée qui n'ont de sens que mis en application dans un contexte.

Outils utilisés

Pour chacune des architectures précédentes, la démarche d'évaluation des performances d'un algorithme peut être segmentée en trois étapes principales :

— la **compilation**, qui permet à partir d'un langage haut niveau de générer du code machine

— la **mesure énergétique**, qui permet de mesurer les performances précises pendant l'exécution du programme

— le **profilage**, qui nous permet de visualiser les performances de cette exécution.

Nous avons regroupé dans le Tableau V.2 les différentes versions des outils utilisées pour chaque type d'architecture :

TABLE V.2 – Outils utilisés pour chaque type d'architecture

	Compilation	**Profilage**	**Mesure énergétique**
CPU Intel	GCC v7.1	GPROF v7.1	CPU Energy Meter
GPU Nvidia	CUDA Toolkit v8.0	Visual Profiler	Nvidia System Management Interface
FPGA Xilinx	SDx 2018.2	SDx 2018.2	Xilinx Power Estimator
FPGA Arria10	Quartus 18.0 IFSO*	Quartus 18.0 IFDPO**	Quartus Prime Power Analyser
FPGA DE1SoC	Quartus 16.0 IFSO*	Quartus 16.0 IFDPO**	Quartus Prime Power Analyser
			*Intel FPGA SDK for OpenCL **Intel FPGA Dynamic Profiler for OpenCL

Démarche de mise en œuvre des résultats et notations

Une des difficultés de mise en œuvre d'une étude comparative entre différentes architectures est de définir un protocole de test cohérent. Dans nos travaux, nous avons défini un cadre dont les grandes lignes sont les suivantes :

— **Compilation :** utilisation du niveau d'optimisation le plus poussé des différents compilateurs (*-03* sur gcc par exemple),

— **Exécution :** 100 lancements indépendants pour chaque version OpenCL.

— **Mesure :** les outils de mesure du temps implémentées ont été effectuées le plus précisément possible, en utilisant par exemple les directives *clock_gettime* POSIX sur CPU pour obtenir la fréquence d'horloge temps réel du système.

— **Résultats :** les résultats présentés sont une moyenne des 100 lancements. Toutefois, sur l'ensemble de nos versions, la dispersion maximale du temps d'exécution par rapport à la moyenne des 100 exécutions est toujours inférieure à 1%.

Même si certaines applications ne sont pas contraintes par leur consommation énergétique, cela pourrait devenir à l'avenir un critère de base pour tous les types d'applications, et non plus seulement pour les systèmes embarqués dans le domaine du HPC basse consommation. C'est pourquoi nous avons mesuré ce critère pour toutes les applications étudiées.

Pour ce qui est des notations, on retrouvera :

1. TEM : temps d'exécution mesuré de l'implémentation.
2. Utilisation empirique : désigne le maximum du pourcentage des différentes ressources utilisées sur le FPGA.
3. Performance relative : correspond au temps d'exécution mesuré rapporté à une utilisation à 100% des ressources.

Chapitre VI

Reconstruction tomographique : accélération d'un opérateur de rétroprojection (Intel)

Sommaire

VI.1 Présentation du cas d'étude et enjeux

VI.1.1 Reconstruction pour la tomographie aux Rayons X

La tomographie aux rayons X [Wang et al., 2008] est un moyen de caractérisation devenu incontournable dans de nombreuses applications d'imageries médicales [Zeng, 2010] (mammographies, imagerie diagnostique ou interventionnelle), de contrôle non destructif (pièces aéronautiques [Chapdelaine, 2019], micro-électronique, ...) ou de sécurité pour le contrôle des bagages et des marchandises à la douane [D'Arcy et al., 2019].

L'objectif principal de la reconstruction tomographie consiste à obtenir une cartographie en 3D d'une grandeur physique de l'objet considéré à partir des mesures 2D d'atténuations à différents angles. Quant il s'agit de densité volumique, on parle alors de tomodensitométrie (« Computed Tomography », CT).

Le principe de la tomodensitométrie est illustré en Figure VI.1. Un objet placé entre une source de rayons X et une matrice de détecteurs tourne autour de l'axe φ. La radiation X émise est atténuée suivant la densité locale de l'objet (ici discrétisé sur trois dimensions), et une matrice de détecteurs sur deux dimensions enregistre l'intensité des rayons reçus pour chaque rotation φ élémentaire.

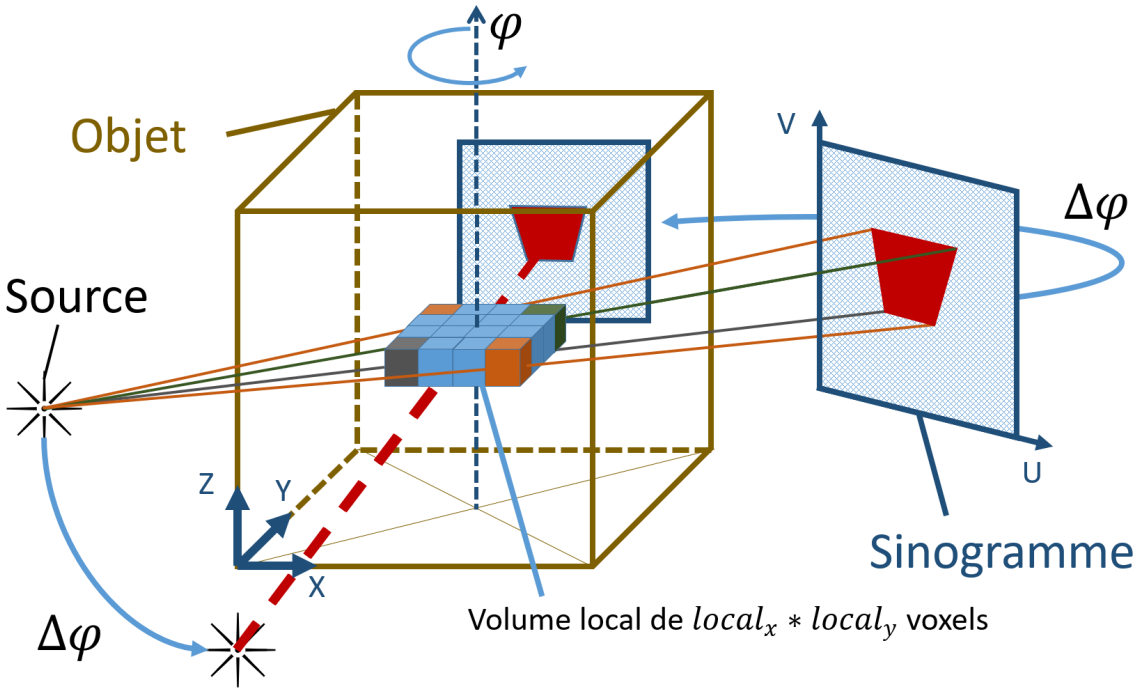


FIGURE VI.1 – Tomodensitométrie : projection 3D.

Ces valeurs sont ensuite enregistrées dans une matrice à trois dimensions suivant (u, v, φ) appelée sinogramme, noté $s_{CT}(u, v, \varphi)$, correspondant à N projections sur un plan de N^2 détecteurs. La reconstruction d'un volume de N^3 **voxels** (pi**XELS VO**lumiques) à partir de ce sinogramme est une tâche gourmande en temps de calcul. Ces dernières décennies, la taille de ce volume n'a fait qu'augmenter, avec actuellement $N = 512$ pour l'imagerie médicale, et $N = 2048$ pour le domaine du contrôle non destructif.

Les algorithmes de reconstruction sont soit analytiques comme la rétro-projection filtrée [Feldkamp et al., 1984], soit itératifs [Geyer et al., 2015]. Ces deux familles de méthodes utilisent un opérateur de rétroprojection qui représente respectivement 90% et 50% du temps de calcul [Gac et al., 2008].

Nous avons donc focalisé notre étude sur l'accélération du rétroprojecteur dont le principe consiste, pour un voxel $\vec{c} = (x, y, z)$ donné de l'objet discrétisé, à additionner la contribution de tous les détecteurs élémentaires (u, v) en ligne avec la source et le voxel considéré pour chaque valeur de φ. On obtient alors la densité $d(\vec{c})$ donnée par la formule VI.1.

$$d(\vec{c}) = \int_0^{2\pi} s_{CT}(u(\varphi, \vec{c}), v(\varphi, \vec{c}), \varphi).w(\varphi, \vec{c})^2 \mathrm{d}\varphi \tag{VI.1}$$

où $w(\varphi, \vec{c}) =$ pondération de distance [Lu et al., 2001],
$(u(\varphi, \vec{c}), v(\varphi, \vec{c})) =$ coordonnées du détecteur dans le sinogramme en ligne avec le voxel $\vec{c}$ et la source.

La distribution des détecteurs étant discrète, l'intégrale se transforme en une somme pour toutes les valeurs de φ. Cet algorithme est particulièrement adapté aux cœurs SIMD, parce que cette somme peut être calculée indépendamment pour chaque voxel de l'objet.

C'est pourquoi les architectures massivement parallèles à base de GPUs représentent, depuis l'apparition de CUDA [Scherl et al., 2007], la très grande majorité des plateformes utilisées aujourd'hui dans ce domaine. Toutefois, quelque soit l'architecture, le principal goulot d'étranglement reste l'accès à la mémoire, la problématique est donc de trouver l'accès pertinent au sinogramme. C'est pourquoi une méthode efficace d'optimisation de ce genre d'algorithmes passe par une réflexion sur les accès contigus en mémoire. Dans le cas de la tomodensitométrie, pour un voxel donné, la récupération des valeurs du sinogramme dessine un motif irrégulier, difficile à prédire. Grouper des accès contigus en mémoire permettra donc d'améliorer de façon performante l'efficacité de l'algorithme.

Dans le domaine de la tomographie, il est nécessaire d'avoir des temps raisonnables de calcul, ce qui suppose de s'intéresser à l'accélération matérielle sur différentes architectures : CPU en multi-cœur (OpenMP, MPI), FPGA, multi-FPGA (imageProX by Siemens [Heigl and Kowarschik, 2007]), mais aussi sur des DSPs, IBM Cell, et les GPUs avant Common Unified Device Architecture (CUDA).

Durant ces dix dernières années, la famille des processeurs graphiques (GPUs) a démontré son efficacité et, plus particulièrement ceux de NVIDIA avec CUDA, ont su se hisser en première place pour le moment incontestée des accélérateurs matériels de ce domaine .

Avec cet essor, les architectures développées sur FPGA en VHDL [Kim et al., 2012, Leeser et al., 2005] ont eu tendance à être mises de côté, parce que d'une part les implémentations correspondantes de rétroprojection nécessitaient un temps de développement bien plus long que celles en CUDA, et d'autre part il persistait certaines limitations concernant la représentation de données (flottant versus fixe).

Notre propos est d'évaluer avec notre méthodologie l'accélération en OpenCL sur FPGA de la reconstruction tomographique. Dans ce domaine, les implémentations en VHDL sur FPGAs ont quasiment disparus [Iain Goddard, 2002, Leeser et al., 2005, Heigl and Kowarschik, 2007, Gac et al., 2008, Kim et al., 2012], mais celles utilisant les outils HLS redeviennent d'actualité [Xu et al., 2010, Ravi et al., 2019] grâce à la plus grande maturité de ces derniers.

VI.1.2 Modèle de référence de l'algorithme de rétroprojection

L'algorithme de référence, qui correspond à notre version initiale (Algo. 10) correspond simplement à l'implémentation sur CPU de l'équation (VI.1) décrite à la section VI.1.1.

Algorithme 10 : Algorithme de référence de l'algorithme de rétroprojection (CPU)

Input : $\alpha[dim_\varphi]$, $\beta[dim_\varphi]$, sinogram$[dim_U * dim_V * dim_\varphi]$
Output : volume, tableau 3D représentant le volume reconstruit

```
1 for ( zn = 0 to dim_Z - 1 ) {
2     for ( yn = 0 to dim_Y - 1 ) {
3         for ( xn = 0 to dim_X - 1 ) {
4             voxel_sum = 0;
5             for ( φ = 0 to dim_φ - 1 ) {
                  /* Calcul des (U_n, V_n) à partir des projections α[φ]
                     et β[φ] */
6                 voxel_sum += sinogram[U_n + V_n * dim_U + φ * dim_U * dim_V];
7             volume[xn,yn,zn] = voxel_sum;
```

Explication de l'algorithme de rétroprojection :

Il s'agit de reconstruire la densité volumique d'un objet discrétisé suivant les trois dimensions. Cette information est enregistrée dans un tableau en trois dimensions, noté *volume*.

Pour chaque voxel élémentaire $\vec{c} = (x, y, z)$, le principe est donc d'additionner la contribution élémentaire de chaque détecteur en ligne avec la source et le voxel pour un φ donné. Pour cela, on effectue une accumulation sur les itérations angulaires (boucle en φ, ligne 5 à 7), et, pour chaque itération, on calcule les coordonnées du détecteur aligné avec le rayon. Nous avons besoin pour cela de coefficients pré-calculés de projection, α et β, qui dépendent uniquement de φ.

Une fois que les coordonnées (U_n, V_n) du détecteur en ligne avec la source et le voxel sont récupérées, il suffit d'ajouter la valeur du sinogramme correspondante dans une variable d'accumulation, ici $voxel_{sum}$ (ligne 6 de l'Algorithme 10). La dernière étape consiste à sauvegarder la variable d'accumulation dans le volume à l'endroit correspondant au voxel en cours, et de continuer les itérations volumiques.

Problématique d'optimisation :

En réalité, les détecteurs ne sont que rarement alignés avec les rayons, et il existe

donc plusieurs méthodes pour calculer la contribution des détecteurs en ligne. La plus précise consiste à effectuer une interpolation bilinéaire sur les coordonnées (U_n, V_n) les plus proches. Une autre méthode plus rapide consiste à récupérer le plus proche voisin, (U_n, V_n) et c'est cette seconde solution que nous avons utilisée dans nos optimisations.

VI.1.3 Analyse de l'algorithme et protocole de test

Protocole de test :
Même si la démarche d'optimisation présentée dans ce chapitre peut s'appliquer quelques que soient les dimensions du problème (taille du sinogramme, nombre de détecteurs), les temps d'exécution présentés en fin de chapitre en dépendent.
Pour clarifier la lecture de la suite de ce chapitre, nous avons donc fixé :
$dim_Z = dim_Y = dim_X = dim_\varphi = dim_U = dim_V = 256$

La première étape d'analyse de l'algorithme présenté ci-dessus nous permet de dresser les observations suivantes :

— Le corps du traitement passe par une quadruple boucle imbriquée (dans l'ordre $z \rightarrow y \rightarrow x \rightarrow \varphi$), dont les dimensions sont ici de 256 itérations chacune. Au total, nous avons donc un algorithme dont un traitement élémentaire sera répété $256^4 = 4 * 1024^3$ fois.

— A chaque itération de la boucle en x, correspond une écriture en mémoire globale.

— A chaque itération en φ, correspondent trois accès en mémoire globale, un pour le sinogramme, et deux pour les coefficients $\alpha[\varphi]$ et $\beta[\varphi]$.

— A chaque itération en φ, sont effectués plusieurs tests conditionnels de débordement de tableau ainsi que le calcul du détecteur qui est le plus proche voisin.

— Également, cette boucle (en φ) a une dépendance de données sous la forme d'un accumulateur à chacune de ses itérations.

— Le sinogramme est un tableau 3D de taille $dim_U * dim_V * dim_\varphi$ Dans notre cas, il contient $256^3 = 16\ 777\ 216$ valeurs flottantes en simple précision, soit une taille en mémoire de 64 Mo.

— Les tableaux des coefficients α et β sont tous les deux de taille dim_φ, soit une taille mémoire de $256 * 4 = 1024$ octets chacun.

Ces observations, nous ont permises de définir les différents leviers d'optimisations qui nous serviront de référence dans le suite de cette section.

Leviers d'optimisation :

Levier 1 : **Tableau volume :** il y a, dans notre cas, 256 fois plus de lectures en mémoire globale que d'écriture, aussi l'accès à ce tableau n'est pas limitant. Toutefois, au niveau de l'interface CPU/FPGA, nous pouvons préciser que ce tableau n'est accessible qu'en écriture depuis le FPGA.
Optimisations envisagées : mémoire globale, qualificatifs write_only/restrict.

Levier 2 : **Tableau sinogramme :** l'accès à ce tableau est le goulot d'étranglement principal, et les optimisations d'accès mémoires dépendent de la régularité de ces derniers.
Optimisations envisagées : mémoire globale ou constante, qualificatifs read_only/const/volatile/restrict.

Levier 3 : **Parallélisme :** nous n'avons ici qu'une seule fonction, que l'on peut difficilement segmenter. Aussi, tous les types de parallélismes peuvent être envisagés ici, à l'exception du parallélisme de fonctions.
Optimisations envisagées : NDRK, réplication de Pipeline Élémentaire, SWIK, déroulage et pipeline de boucles.

Levier 4 : **Tableaux α et β :** au vu de la faible taille de ces tableaux et de leur forte régularité, il est judicieux de les implémenter dans une mémoire rapide.
Optimisations envisagées : mémoire constante ou locale, déroulage ou pipeline de la boucle de copie.

Levier 5 : **Optimisations fines :** les différents tests de débordement de tableau à l'intérieur de la boucle la plus imbriquée doivent être correctement équilibrés.
Optimisations envisagées : Réécriture des branchements conditionnels, fusion des boucles, optimisations spécifiques à l'algorithme.

Le Tableau VI.1 synthétise ces leviers d'optimisations par catégories, et notre démarche d'optimisation consiste à y choisir les plus pertinentes.

TABLE VI.1 – Les différents leviers d'optimisations pertinents

Tableau volume	**Tableau sinogramme**	**Type de parallélisme**	**Tableaux α et β**	**Optimisations fines**
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only (O/N*), restrict (O/N)	read_only (O/N), restrict (O/N), volatile (O/N)	Déroulage ou pipeline	Déroulage ou pipeline	(O/N)
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale ou Constante	SWIK ou NDRK ou réplication PE	Locale ou Constante	Équilibrage
*O/N :	Oui/Non			

VI.2 Exploration des optimisations FPGA

Dans cette section nous décrivons le chemin d'optimisation parcouru sur l'Arria 10, et expliquons nos choix à chaque étape. En nous basant sur les observations effectuées à la sous-section VI.1.3 et sur les prérequis établis au chapitre III, nous pouvons conjecturer

les optimisations les plus efficaces. Ce choix effectué parmi les optimisations possibles présentées au Tableau VI.1, est mis en évidence dans le Tableau VI.2.

TABLE VI.2 – Hypothèses d'optimisations pertinentes

Tableau volume	**Tableau sinogramme**	**Type de parallélisme**	**Tableaux α et β**	**Optimisations fines**
Politique d'accès	***Politique d'accès***	***Boucles x, y, z, φ***	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict	Déroulage φ, Pipeline x, y, z	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	SWIK	Locale	Équilibrage

Dans les sous-sections suivantes, nous présentons tout d'abord notre première implémentation OpenCL, à partir de laquelle nous appliquons les cinq leviers d'optimisations (Tableau VI.1), ce qui restreindra au fur et à mesure le champ d'exploration. A chaque étape, une optimisation validée sera marquée en vert dans la colonne du tableau correspondant au levier d'optimisation exploré.

VI.2.1 Implémentation OpenCL : version initiale et levier 1

Notre première implémentation consiste à réécrire le code initial (en C) en l'adaptant à la syntaxe OpenCL.

En plus de cela, les qualificatifs *write_only* et *read_only* pour, respectivement, le *volume* et le *sinogramme* (*Leviers 1 et 2*), ainsi que le mot clef *restrict* sont directement mis en œuvre dès cette première version, puisqu'il est également certain que ces optimisations, au vu de la politique d'accès à ces ressources, sont pertinentes.

Les différents constructeurs recommandent, pour l'optimisation du parallélisme (*Levier 3*), d'implémenter des SWIKs, et c'est donc l'optimisation choisie pour cette première version. Nous choisissons pour le moment d'implémenter les boucles (z, y, x, φ) en pipeline.

Au vu de la taille des tableaux α et β, il est logique de les implémenter en mémoire locale, avec déroulage de la boucle de copie de ces données car cela reste en théorie le plus optimal (*Levier 4*).

Nous avons également choisi d'implémenter l'équilibrage des tests conditionnels (*Levier 5*) car cette optimisation utilise moins de ressources tout en permettant une meilleure prédiction de la régularité d'accès au sinogramme,

On obtient alors comme état initial d'optimisation celui présenté au tableau VI.3.

Certaines optimisations choisies pour cet état initial sont évalués dans les sous-sections suivantes, et permettent ainsi de confirmer la validité de nos choix initiaux et, dans le cas contraire, de les modifier.

TABLE VI.3 – État initial des optimisations

Tableau volume	Tableau sinogramme	Type de parallélisme	Tableaux α et β	Optimisations fines
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict	Pipeline	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	**A définir**	SWIK	Locale	Équilibrage

VI.2.2 Optimisation de l'accès au sinogramme (Levier 2)

A partir de l'état initial évoqué à la section précédente, nous voulons dorénavant trouver quels autres qualificatifs et zones mémoires seraient les plus pertinents pour le tableau du sinogramme.

Afin d'optimiser les accès au tableau sinogramme, nous avons :

— comparé les différentes zones mémoires possibles sur le FPGA en OpenCL,

— implémenté un mécanisme de pré-chargement en mémoire suivant une démarche d'adéquation algorithme architecture, inspirée des travaux de [Gac et al., 2008] tirant parti d'un mécanisme de mise en cache 3D prédictif du sinogramme [Mancini and Eveno, 2004].

Dans les deux sous-sections suivantes, ces différentes approches sont présentées. La troisième sous-section expose leurs résultats respectifs .

VI.2.2.1 Choix parmi les zones mémoires existantes

Nous avons, comme évoqué dans notre méthodologie à la section IV.3, plusieurs choix possibles pour stocker cette variable en fonction que les accès soient :

— très réguliers : (*T1_V1*),

— assez réguliers : mémoire globale et pas de qualificatif *volatile* (*T1_V2*),

— pas ou très peu réguliers : mémoire globale et qualificatif *volatile* (*T1_V3*).

En plus de ces trois options, il est également possible d'implémenter manuellement une méthode de pré-chargement en mémoire des données du sinogramme, que nous présentons à la section suivante.

VI.2.2.2 Implémentation manuelle d'un cache

Nous allons donc tirer parti de la mémoire locale, et essayer de partager au mieux les données du sinogramme entre chaque itérations successives *T1_V4*). Le goulot d'étranglement le plus important de notre algorithme correspond à l'accès au sinogramme. Pour optimiser cet accès, nous avons implémenté un mécanisme qui pré-charge intelligemment certaines de ces données en mémoire.

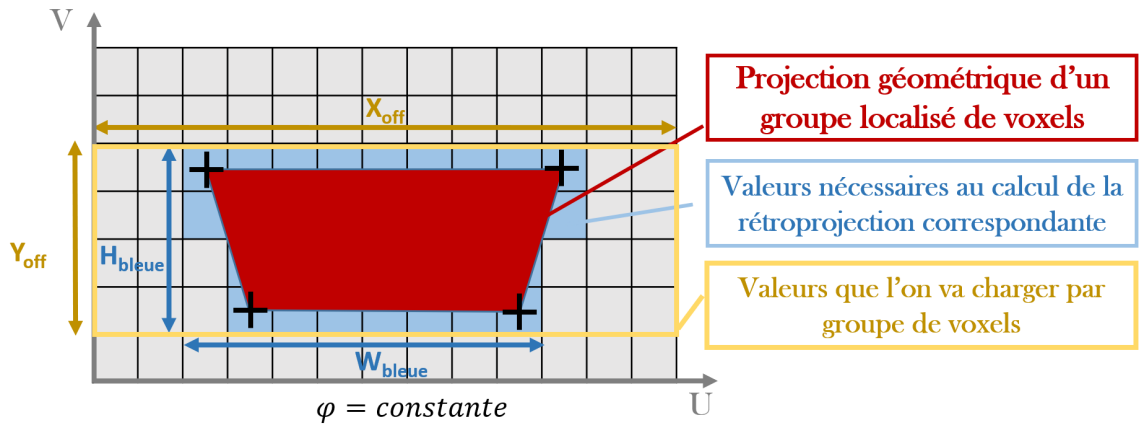


FIGURE VI.2 – Pré-chargement optimisé du sinogramme pour un groupe de voxels.

Comme expliqué à la Section VI.1.1, (U, V, φ) représente la projection en ligne d'un voxel sur la matrice des détecteurs. Pour un φ donné, une projection d'un groupe de $(local_x, local_y, 1)$ voxels localisé sur le plan $(0, \vec{x}, \vec{y})$ ressemble à la projection géométrique illustrée aux Figures VI.1 et VI.2. Sur cette dernière figure, les quatre croix noires correspondent aux projections des quatre coins du rectangle localisé de voxels.

Ainsi, pour calculer les densités correspondant à une zone localisée de voxels, le programme doit accéder à toutes les cases du sinogramme qui sont autour de la projection géométrique (*Valeurs nécessaires au calcul*). Le motif de cette projection n'est pas régulier d'un groupe localisé de voxels à un autre, et le compilateur d'Intel ne peut prédire d'une itération à l'autre où aller chercher en mémoire les bonnes valeurs. Dans (VI.2), on note W_{bleue} et H_{bleue} la largeur et la hauteur de la zone autour de la projection géométrique. Ces formules sont garanties par la géométrie de la rétroprojection 3D.

$$W_{bleue} < \sqrt{local_x^2 + local_y^2},\ H_{blue} <= 4 \tag{VI.2}$$

Cette équation est vraie quelque soit le groupe de voxels considéré. A partir de ce constat, et en choisissant la taille du rectangle de voxels avec $local_z = local_y = 1$ et $local_x = 256$, on sait qu'une matrice de dimension $(X_{off}, Y_{off}) = (256, 4)$ est suffisamment grande pour contenir la projection géométrique du groupe localisé de voxels. Ce tableau (*valeurs que l'on va charger)*, représenté sur la Figure VI.2, contient plus de valeurs que nécessaire, mais sa régularité et sa faible taille permettent de le charger en mémoire locale afin de les partager entre 256 calculs de voxels.

Ainsi, l'algorithme de pré-chargement en mémoire (Algo. 11) va s'appuyer sur cette remarque. Dans notre implémentation, chaque work-group est constitué de 16*16 work-item, pour correspondre à un découpage d'un groupe localisé de voxels. Pour un groupe de voxels donné, la première étape va être de calculer les projections des voxels aux extrémités du rectangle, parce que leur projection correspond aux extrémités de la zone de projection géométrique. A partir de ces quatre valeurs, nous allons chercher la zone (X_{off}, Y_{off}) correspondante, puis répartir le pré-chargement des valeurs du sinogramme entre les différents work-items. Après une synchronisation, il suffit d'aller récupérer comme lors des versions précédentes les valeurs adéquates du sinogramme, qui est

Algorithme 11 : Optimisation du pré-chargement des données

Input : $\alpha[dim_\varphi]$, $\beta[dim_\varphi]$, sinogram[$dim_U * dim_V * dim_\varphi$]
Output : volume, tableau 3D représentant le volume reconstruit

```
1  __attribute__((reqd_work_group_size(1,1,1)))
2  __kernel void backprojection3D /* Arguments                                  */
3  {
4   /* Copie en mémoire locale de α et β                                        */
5  local int local_sinogram[dim_U * 4];
6   for ( zn = 0 to dim_Z ) {
7       for ( yn = 0 to dim_Y ) {
            /* Calcul des extrémités de la zone et chargement dans
               local_sinogram                                                    */
8           for ( xn = 0 to dim_X ) {
9               voxel_sum = 0;
10              for ( φ = 0 to dim_φ ) {
11                  /* Calcul du plus proche voisin (U_n V_n) à l'aide de
                       α[φ] et β[φ]                                              */
12                  voxel_sum += sinogram[U_n + V_n * dim_U + φ * dim_U * dim_V];
13              volume[xn,yn,zn] = voxel_sum;
14 }
```

cette fois ci en mémoire locale.

Le principal désavantage de cet algorithme est que l'on va récupérer plus de cases mémoires que nécessaire, mais cet inconvénient est minime en comparaison des avantages majeurs qu'apportent le pré-chargement, qui permet de disposer :

— d'une zone mémoire de taille identique quels que soient les groupes de voxels considérés,

— d'un accès mémoire contigu pour le sinogramme du fait de la mutualisation du chargement des voxels.

VI.2.2.3 Résultats et choix

Pour l'exploration de ce levier d'optimisation, nous avons donc implémenté les quatre solutions précédentes, et le tableau VI.4 résume leurs différents temps d'exécution, ainsi que leurs consommations en ressources sur la carte Arria 10.

Les résultats obtenus pour les versions *T1_V1*, *T1_V2*, et *T1_V3* sont cohérents avec les spécificités des différentes zones mémoire explorées. En effet, le motif d'accès au sinogramme est pseudo-régulier, avec une réutilisation des données d'environ 40% d'une itération sur la boucle x à l'autre. Il est donc logique que la zone mémoire la plus pertinente soit un compromis entre absence de cache et cache optimisé pour les succès de cache.

En ce qui concerne la mise en cache manuelle (*T1_V4*), qui se repose sur l'analyse spécifique de l'algorithme considéré, on peut observer que le temps d'exécution mesuré

TABLE VI.4 – Choix de la zone mémoire du sinogramme : performances des implémentations (Arria 10)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (s)	Performance relative (s)
T1_V1 : mémoire constante	24	197.92	1666.45	399.95
T1_V2 : mémoire globale non volatile	24	196.97	124.82	29.96
T1_V3 : mémoire globale volatile	14	198.54	951.24	133.17
T1_V4 : mémoire globale volatile et cache manuel	28	197.57	158.47	44.37

est bien plus rapide que celui des versions *T1_V1* et *T1_V3*, mais reste toutefois plus lent que celui de la version *T1_V2*.

La zone mémoire la plus optimale pour stocker le sinogramme sur le FPGA est donc la mémoire globale non volatile. Les outils d'Intel implémentent alors un cache qui pénalise peu les défauts de cache, tout en permettant, le cas échéant, d'accélérer significativement les succès de cache.

On obtient alors le nouvel état d'optimisation, présenté au Tableau VI.5.

TABLE VI.5 – État des optimisations après choix de la zone mémoire du sinogramme

Tableau volume	Tableau sinogramme	Type de parallélisme	Tableaux α et β	Optimisations fines
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, **non volatile**	Pipeline	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	**Globale**	SWIK	Locale	Équilibrage

VI.2.3 Type de parallélisme (Levier 3)

Le choix précédent de la localisation mémoire du sinogramme est indépendant du choix du parallélisme, et c'est pourquoi nous l'avons effectué en premier. Maintenant que la zone mémoire la plus efficace a été identifiée, il s'agit ici de choisir quel est le type de parallélisation le plus pertinent pour ce kernel.

La première étape de parallélisation est de choisir entre un NDRK et un SWIK. En effet, ce choix va conditionner le reste des optimisations comme la réplication des cœurs de calcul, la vectorisation, le parallélisme ou le pipeline de boucles.

Le parallélisme de notre algorithme pouvant être exprimé par le parallélisme de tâches et/ou de données. Il convient alors d'explorer les deux approches simultanément.

Pour cela, nous repartons de l'état défini lors des optimisations précédentes, présenté au Tableau VI.6, et cherchons à définir quels sont les types de parallélismes pertinents dans notre cas de figure.

TABLE VI.6 – État des optimisations avant parallélisme.

Tableau volume	**Tableau sinogramme**	**Type de parallélisme**	**Tableaux α et β**	**Optimisations fines**
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, non volatile	**A définir**	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	**A définir**	Locale	Équilibrage

VI.2.3.1 Approche SWIK

Les FPGAs se prêtent mieux au parallélisme de tâches et de fonctions qu'au parallélisme de données massif. En effet, même si les mémoires internes des FPGAs récents sont rapides, elles ne peuvent rivaliser, pour le moment, avec la vitesse d'approvisionnement en ressources des GPUs qui possèdent un nombre bien plus important de bus de plus grande largeur. Pour pallier à cet inconvénient, nous allons explorer l'approche SWIK.

Choisir d'exprimer notre algorithme en SWIK conditionne et réduit les possibilités du parallélisme, dans notre cas de figure, au pipeline et au déroulage de boucles.

L'algorithme de la version *T1_V2* contenant 4 boucles imbriquées, il convient de s'intéresser à la plus profonde, pour obtenir une granularité la plus fine possible. Dans notre cas, il s'agit de la boucle en φ.

Pour vérifier la pertinence de notre choix, nous avons également implémenté une version où la boucle en x est déroulée.

Aussi, nous avons implémenté, en partant de l'état présenté au Tableau VI.6 les versions suivantes :

— Pipeline de la boucle en φ (*T1_V2*)

— Déroulage de la boucle en φ x16 (*T1_V5*), x32 (*T1_V6*), x40 (*T1_V7*)

— Déroulage de la boucle en x x16 (*T1_V8*)

En réalité, une compilation rapide des codes OpenCL des différentes versions nous permet de savoir que 40 est le facteur de déroulage maximum possible pour que notre implémentation rentre sur le FPGA.

La boucle en φ ayant 256 itérations, il est fortement recommandé que le facteur de déroulage divise le nombre d'itérations totales. Aussi, le facteur optimal maximal pressenti

est 32 (*T1_V6*), mais nous avons tout de même implémenté les versions *T1_V5*, et *T1_V7* afin de valider cette intuition.

On retrouve toutes ces implémentations au Tableau VI.7, et il convient de préciser que l'optimisation intégrant le pipeline de la boucle en φ correspond à la version *T1_V2* qui a déjà été implémentée à l'étape précédente.

TABLE VI.7 – Implémentations SWIK : performances (Arria 10)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (s)	Performance relative (s)
T1_V2 : Pipeline φ	24	196.97	124.82	29.95
T1_V5 : Déroulage φ x16	50	170.45	19.09	9.54
T1_V6 : Déroulage φ x32	62	150	5.34	3.31
T1_V7 : Déroulage φ x40	80	134.94	68.63	54.90
T1_V8 : Déroulage x x16	87	142.41	29.12	25.33

De ces résultats ressort clairement l'optimisation la plus efficace : la version *T1_V6*. En effet, les versions *T1_V2* et *T1_V5* sous-utilisent les ressources disponibles sur la carte, et ne tirent donc pas complètement parti du potentiel parallélisme de l'architecture. Une version non présentée ici, qui inclut le déroulage de la boucle en φ d'un facteur 64 consomme quant à elle trop de ressources pour être implémentée sur notre architecture.

Le facteur de réplication maximal possible sur cette architecture est pour notre algorithme de 40. Toutefois, ce facteur n'étant pas un diviseur du nombre d'itérations global, les performances, comme on pouvait s'y attendre, sont sous-optimales.

Quant à l'optimisation *T1_V8*, le choix de dérouler une boucle qui n'est pas la plus imbriquée augmente les ressources à dupliquer, telles que les variables ou encore les calculs inclus dans cette boucle. Cette optimisation n'est pas efficace dans ce cas, car la granularité n'est pas assez fine. En effet, non seulement l'empreinte logique est plus importante que de dérouler la boucle en φ (87% contre 50%) pour le même facteur de déroulage (x16), mais de plus, nous avons un temps d'exécution plus important.

Pour cet algorithme, le parallélisme le plus efficace pour un SWIK est donc de dérouler la boucle en φ d'un facteur 32.

Nous n'avons pas évoqué ici le parallélisme de fonctions, puisque notre algorithme est composé d'une seule fonction complexe qui est difficilement fragmentable.

VI.2.3.2 Approche NDRK

Le modèle d'implémentation NDRK a l'avantage de partager entre différents work-items des objets mémoires. Dans notre cas d'étude, le volume 3D est exploré par itérations successives suivant les trois dimensions, et nous choisissons donc d'affecter un work-item à chaque voxel élémentaire (x, y, z).

Des trois boucles en z, y, x, celle la plus imbriquée, et dont les itérations successives auront certaines valeurs en commun, est la boucle en x, d'où le choix de work-groups de taille $(dim_Z, dim_Y, dim_X) = (1, 1, 256)$ pour mutualiser les ressources locales de la boucle en x.

Utiliser un kernel de type NDRK sur trois dimensions permet donc de masquer les trois premières boucles, en laissant la gestion de celles-ci à l'outil qui va mettre en place une pile d'exécution des différents work-items. Chacun d'eux doit alors gérer une boucle en φ comme présenté à l'Algo. 12.

Algorithme 12 : Algorithme simplifié illustrant une implémentation NDRK

Input : $\alpha[dim_\varphi]$, $\beta[dim_\varphi]$, sinogram[$dim_U * dim_V * dim_\varphi$]
Output : volume, tableau 3D représentant le volume reconstruit

```
1  __attribute__((num_compute_units(A)))
2  __attribute__((num_simd_work_items(B)))
3  __attribute__((reqd_work_group_size(1,1,256)))
4  __kernel void backprojection3D /* Arguments                          */
5  {
6   /* Initialisation variables et copie locale                         */
7   /* Récupération des identifiants du work-item                       */
8   xn = get_global_id(0);
9   yn = get_global_id(1);
10  zn = get_global_id(2);
11  voxel_sum = 0; /* Accumulateur                                      */
12  #pragma unroll C
13  for ( φ = 0 to dim_φ ) {
14  |   /* Calcul du plus proche voisin (U_n V_n) à l'aide de α[φ] et
    |      β[φ]                                                         */
15  |   voxel_sum += sinogram[U_n + V_n * dim_U + φ * dim_U * dim_V];
16  volume[xn,yn,zn] = voxel_sum;
17 }
```

Les types de parallélisme les plus efficaces dans ce cas sont la vectorisation des work-items ainsi que la réplication des pipelines élémentaires. Néanmoins, comme chacun des work-items doit exécuter une boucle sur les itérations angulaires, il est possible de dérouler ou de pipeliner cette boucle.

Nous avons donc implémenté plusieurs versions :

— Réplication du pipeline élémentaire x2, déroulage de la boucle en φ x16 (*N1_V1*),

— Réplication du pipeline élémentaire x16, pipeline de la boucle en φ (*N1_V2*),

— Réplication du pipeline élémentaire x16 et vectorisation x4, pipeline de la boucle en φ (*N1_V3*).

On retrouve au Tableau VI.8 les résultats de ces différentes itérations, que l'on compare avec la meilleure implémentation SWIK que l'on a obtenue pour le moment.

Du point de vue de l'utilisation logique, l'optimisation la plus efficace est la *N1_V1*, et cela s'explique par le déroulage de la boucle en φ qui, étant la boucle la plus imbriquée, réduit les ressources à répliquer et donc l'empreinte mémoire de l'implémentation par rapport à un parallélisme de granularité moins fine, comme la réplication du pipeline élémentaire.

TABLE VI.8 – Implémentations NDRK et comparaison SWIK : performances (Arria 10)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (s)	Performance relative (s)
SWIK				
T1_V6 : Déroulage φ x32	62	150	5.34	3.31
NDRK				
N1_V1 : Réplication PE x2, déroulage φ x16,	71	162.04	42.35	30.07
N1_V2 : Réplication PE x16, φ en pipeline	83	168.23	39.62	32.89
N1_V3 : Réplication PE x16, φ en pipeline, Vectorisation x4	83	165.18	39.76	33.00

On observe également que la vectorisation est ici contre productive. En effet, cette optimisation impose de plus fortes contraintes sur la synchronisation des flots de données, ce qui abaisse légèrement la fréquence du kernel, et augmente légèrement le temps d'exécution mesuré.

En ce qui concerne les implémentations NDRK, la plus optimale consiste donc à répliquer le pipeline élémentaire pour permettre ensuite à l'outil OpenCL d'alimenter en works-groups indépendant ces différents cœurs de calcul, mais cette version reste toutefois moins efficace tant en temps d'exécution qu'en utilisation logique que la version SWIK (*T1_V6*).

VI.2.3.3 Résultats et choix

Dans cette section, nous avons commencé par exprimer notre algorithme en SWIK, tout en tirant parti du déroulage des différentes boucles. Il en est ressorti que la meilleure solution était d'exprimer le parallélisme au plus près des calculs élémentaires, c'est à dire en déroulant la boucle en φ. Ensuite, en exprimant notre rétroprojection en NDRK, nous avons montré que la solution offrant un meilleur temps d'exécution correspondait à la réplication complète du pipeline élémentaire. Cela s'explique par le fait que l'expression en work-items et work-groups d'un programme OpenCL permet une répartition efficace de ces derniers sur différents pipelines.

En réalité, dans notre algorithme, même s'il est possible d'exprimer un parallélisme de données sur les triplets (x, y, z), le cœur des calculs est composé de la boucle en φ qui accumule, dans une variable, la contribution élémentaire des détecteurs en ligne avec la source, pour différentes variations angulaires. Chacune de ces itérations a besoin d'une valeur différente des tableaux α et β, et il a été montré que les mécanismes de caches, dans le cas d'un déroulage de cette boucle, s'avèrent efficace pour partager les ressources communes du sinogramme. Cela explique pourquoi l'approche SWIK *T1_V6*, où les boucles (x, y, z) sont en pipeline et la boucle en φ est déroulée d'un facteur 32, est l'implémentation la plus rapide parmi les différentes parallélisations possibles.

Nous avons donc choisi cette implémentation, et l'état des optimisations à l'issue de cette étape est résumé au Tableau VI.9.

TABLE VI.9 – État des optimisations après parallélisme.

Tableau volume	**Tableau sinogramme**	**Type de parallélisme**	**Tableaux α et β**	**Optimisations fines**
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, non volatile	**Pipeline** x, y, z **Déroulage** φ **x32**	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	**SWIK**	Locale	Équilibrage

VI.2.4 Accès aux tableaux α et β (Levier 4)

Au début de notre parcours d'optimisation, nous avions convenu qu'au vu de la taille des tableaux α et β, il paraissait logique de les implémenter en mémoire locale. Cette section d'optimisation est donc consacrée à évaluer la manière dont on les copie depuis la mémoire globale du FPGA ainsi que la meilleure zone mémoire pour ces tableaux. L'état des optimisations avant cette étape est résumé au Tableau VI.10.

TABLE VI.10 – État des optimisations avant évaluation tableaux α et β.

Tableau volume	**Tableau sinogramme**	**Type de parallélisme**	**Tableaux α et β**	**Optimisations fines**
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, non volatile	Pipeline x, y, z Déroulage φ x32	**A définir**	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	SWIK	**A définir**	Équilibrage

VI.2.4.1 Politique de copie

Copier un objet de la mémoire globale du FPGA à la mémoire locale se fait à l'aide d'une boucle, et il existe donc deux manières de les optimiser : à l'aide d'un déroulage (*T1_V6*) ou d'un pipeline de boucle (*T1_V9*).

Si le pipeline permet d'avoir un temps d'initialisation réduit sur la boucle de copie, il est possible que le déroulage de boucle soit intéressant eu égard aux performances

brutes, car il permet de paralléliser par groupe la copie des données. Toutefois, il faut veiller à ce que l'empreinte mémoire ne soit pas trop importante, et par souci de clarté, les résultats de ces optimisations sont présentés au Tableau VI.11 de la section VI.2.4.3 avec les résultats de la sous-section suivante.

VI.2.4.2 Structure mémoire

En ce qui concerne la localisation en mémoire de ces deux tableaux, plusieurs choix sont en réalité possibles. On retrouve l'implémentation en mémoire locale (*T1_V6*), celle en mémoire constante qui permet d'implémenter des caches efficaces (*T1_V10*), mais nous avons également implémenté un registre à décalage, qui permet de générer un flot de données à partir des deux tableaux pour alimenter efficacement les cœurs de calcul (*T1_V11*). Ce concept est illustré en Figure VI.3.

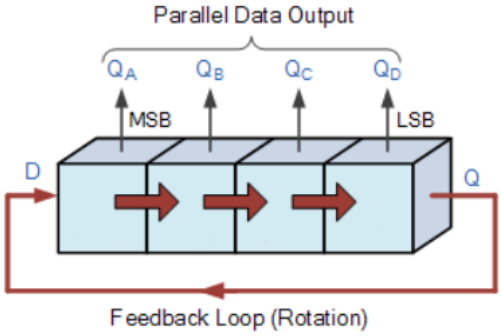


FIGURE VI.3 – Registre à décalage

Source: electronics-tutorials.ws

Si un algorithme accède à un objet mémoire, et, qu'à chaque itération on a besoin de la case mémoire suivante, il est possible d'implémenter un registre à décalage. Ainsi, la donnée est décalée d'un cran à chaque coup d'horloge, et la donnée de la dernière case du tableau est copiée dans la première case, créant une boucle complète. De plus, il est possible d'accéder en parallèle à certaines cases du tableau, si tant est que les ressources disponibles permettent d'implémenter ce mécanisme.

Les avantages de cette solution sont de réduire l'empreinte en ressources mémoires de ces objets et d'optimiser leurs latences d'accès.

Si l'on reprend l'algorithme de référence (Algo. 10, section VI.1.2), nous avions pu voir que, pour chaque itération élémentaire (x, y, z, φ), on accédait aux valeurs $\alpha[\varphi]$ et $\beta[\varphi]$ en boucle, et cet accès se prête bien à l'implémentation d'un registre à décalage, qui est détaillé à l'Algo. 13. L'implémentation se fait aux lignes 6,7 (initialisation),14,15, et 16. Même si cette écriture ne paraît pas intuitive du point de vue d'un programmeur, l'outil va la reconnaître et implémenter le registre à décalage.

VI.2.4.3 Résultats et choix

Dans les deux sous-sections précédentes, nous avons donc trois nouvelles optimisations, portant tant sur la copie des tableaux α et β que sur la manière dont ceux-ci sont

Algorithme 13 : Implémentation d'un registre à décalage pour les tableaux α et β

Input : $\alpha[dim_\varphi]$, $\beta[dim_\varphi]$, sinogram[$dim_U * dim_V * dim_\varphi$]
Output : volume, tableau 3D représentant le volume reconstruit

```
1  __attribute__((reqd_work_group_size(1,1,1)))
2  __kernel void backprojection3D /* Arguments                          */
3  {
4   local int2 RD[dim_φ];
5   #pragma unroll ; /* Déroule toute la boucle                         */
6   for ( φ = 0 to dim_φ ) {
7   |   RD[φ] = (α[φ], β[φ]);
8   for ( zn = 0 to dim_Z ) {
9   |   for ( yn = 0 to dim_Y ) {
10  |   |   for ( xn = 0 to dim_X ) {
11  |   |   |   voxel_sum = 0;
12  |   |   |   #pragma unroll 32;
13  |   |   |   for ( φ = 0 to dim_φ ) {
14  |   |   |   |   RD[dim_φ - 1] = RD[0];
15  |   |   |   |   for ( i = 0 to dim_φ - 2 ) {
16  |   |   |   |   |   RD[i] = RD[i+1];
17  |   |   |   |   /* Calcul du plus proche voisin (U_n V_n) à l'aide de
    |   |   |   |      RD[φ]                                              */
18  |   |   |   |   voxel_sum += sinogram[U_n + V_n * dim_U + φ * dim_U * dim_V];
19  |   |   |   volume[xn,yn,zn] = voxel_sum;
20 }
```

stockés. En résumé, nous avons :

- Copie avec une boucle déroulée (*T1_V6*)
- Copie avec une boucle en pipeline (*T1_V9*)
- Stockage en mémoire constante (*T1_V10*)
- Structure en registre à décalage (*T1_V11*)

Le Tableau VI.11 synthétise les différentes versions implémentées pour choisir les bons paramètres pour α et β .

En ce qui concerne la copie du tableau, on remarque que la version déroulée a une empreinte mémoire plus faible que si elle est en pipeline. En réalité, même si dans la plupart des cas, dérouler une boucle équivaut à répliquer l'architecture de la boucle correspondante, dans ce cas, il y a réplication des bus de lecture afin de copier de manière contiguë les 512 valeurs des Tableaux α et β efficacement. La copie des tableaux entre la mémoire globale et locale est donc systématiquement déroulée.

Quant à la localisation de ces tableaux, la mémoire locale reste la meilleure solution, tant en terme de l'utilisation logique qu'au niveau du temps d'exécution. Le kernel avec mémoire constante, bien qu'ayant une fréquence plus élevée, reste pénalisée par des

TABLE VI.11 – Optimisations des tableaux α et β : performances (Arria 10)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (s)	Performance relative (s)
Boucle de copie de α et β				
T1_V6 : Déroulage	62	150	5.34	3.31
T1_V9 : Pipeline	65	150.55	5.53	3.59
Localisation de α et β				
T1_V6 : Mémoire locale	62	150	5.34	3.31
T1_V10 : Mémoire constante	64	152.34	5.54	3.54
T1_V11 : Registre à décalage	63	140.15	5.49	3.46

caches efficaces mais moins rapide que la mémoire locale, même avec copie de ces derniers.

La version incluant le registre à décalage quant à elle à une fréquence assez basse par rapport aux deux autres implémentations, qui s'explique par la nécessité de synchroniser efficacement le décalage des valeurs en plus de gérer le déroulage de la boucle en φ 32 fois.

Notre choix initial d'implémenter les tableaux α et β en mémoire locale avec déroulage de la boucle de copie est confirmé par les résultats obtenus, et le nouvel état d'optimisation est présenté au Tableau VI.12.

TABLE VI.12 – Choix des paramètres optimaux (tableaux α et β)

Tableau volume	Tableau sinogramme	Type de parallélisme	Tableaux α et β	Optimisations fines
Politique d'accès	***Politique d'accès***	***Boucles x, y, z, φ***	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, non volatile	Pipeline x, y, z Déroulage φ x32	**Déroulage**	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	SWIK	**Locale**	Équilibrage

VI.2.5 Optimisations fines (Levier 5)

La plupart des optimisations usuelles ayant été abordées, il nous reste désormais à vérifier la pertinence de deux optimisations fines de notre algorithme implémentées dès la première version OpenCL, à savoir l'équilibrage des tests conditionnels et la fusion des boucles.

VI.2.5.1 Fusion des boucles

En s'appuyant sur nos observations (section IV.1.2.4 du Chapitre IV), nous avions fait le choix de ne pas fusionner les boucles, du fait du très grand nombre d'itérations. A partir de l'état des optimisations présenté au Tableau VI.12, nous avons néanmoins voulu vérifier que ce paramètre ne nous fait en effet pas gagner en performances.

Nous avons donc implémenté un kernel presque identique au *T1_V6* en rajoutant la fusion des trois boucles imbriquées (x, y, z) (*T1_V12)*, dont nous retrouverons le nouvel état d'optimisation au Tableau VI.13.

VI.2.5.2 Équilibrage des test conditionnels

La version de référence de cet algorithme (Algo. 10) doit, pour le calcul du plus proche voisin, effectuer un certain nombre de tests conditionnels, détaillés des lignes 2 à 9 de l'Algo 14.

Cet algorithme, sur CPU, ne pose pas de problème particulier, mais sur FPGA, où l'architecture est créée à partir de la définition algorithmique, les lignes 3, 6, et 9 sont toutes les trois traduites en ressources matérielles différentes, et nous nous retrouvons avec trois accès à la mémoire globale à différents endroits de notre algorithme, ce qui est coûteux en ressources logiques.

C'est pourquoi, dès la première itération de notre algorithme en OpenCL (lignes 11 à 22), nous avons rajouté la variable $sinogram\_coeff$. Celle-ci sert d'indicateur, et sa valeur est mise à 1 dans les cas où il y avait dans la version CPU un accès au sinogramme. Une fois tous les tests conditionnels parcourus, nous implémentons un unique accès au sinogramme (ligne 22), ce qui économise potentiellement des ressources.

Certes, l'accès au sinogramme est dorénavant effectué pour chaque itération de φ, mais cette régularité améliore la prévision d'accès à cette donnée par l'outil.

VI.2.5.3 Résultats et choix

Le Tableau VI.13 présente les performances des optimisations fines évoquées précédemment.

Notre choix initial de ne pas fusionner les boucles mais d'équilibrer les tests conditionnels est l'implémentation la plus optimale. En ce qui concerne l'équilibrage des tests, l'implémentation *T1_V13* consomme plus de ressources, et le temps d'exécution mesuré est plus important que pour la version *T1_V6*. La fusion des boucles quant à elle est très légèrement moins efficace que l'implémentation la plus optimale, malgré une fréquence légèrement plus rapide.

De ces observations, nous obtenons le Tableau VI.14, qui représente l'état final des optimisations de notre algorithme.

VI.3 Bilan

Dans ce bilan, nous commençons par résumer notre parcours d'optimisation (section VI.3.1) en revenant sur les différentes performances obtenues sur FPGA en OpenCL, puis

Algorithme 14 : Équilibrage de l'accès au sinogramme dans les tests conditionnels

1 **Implémentation initiale :**

2 **if** *condition1_un_vn* **then**
  /* Calculs */
3 voxel_sum += $sinogram[U_n + V_n * dim_U + \varphi * dim_U * dim_V]$;
4 **else if** *condition2_un* **then**
  /* Calculs */
5 **if** *condition3_vn* **then**
6 voxel_sum += $sinogram[U_n + \varphi * dim_U * dim_V]$;
7 **else if** *condition4_vn* **then**
  /* Calculs */
8 **if** *condition5_un* **then**
9 voxel_sum += $sinogram[V_n * dim_U + \varphi * dim_U * dim_V]$;

10 **Implémentation de l'équilibrage :**

11 sinogram_coeff = 0;
12 **if** *condition1_un_vn* **then**
  /* Calculs */
13 sinogram_coeff = 1;
14 **else if** *condition2_un* **then**
  /* Calculs */
15 **if** *condition3_vn* **then**
16 V_n = 0;
17 sinogram_coeff = 1;
18 **else if** *condition4_vn* **then**
  /* Calculs */
19 **if** *condition5_un* **then**
20 U_n = 0;
21 sinogram_coeff = 1;
22 voxel_sum += $sinogram[U_n + V_n * dim_U + \varphi * dim_U * dim_V]$ * sinogram_coeff;

nous comparons à la section VI.3.2 la meilleure optimisation obtenue à d'autres optimisations sur CPU (OpenMP), GPU (CUDA), et FPGA (VHDL). Finalement, nous concluons sur l'efficacité des outils OpenCL d'Intel pour l'accélération de cet algorithme, et dressons un bilan sur la possible résurgence des architectures FPGAs dans le domaine de la tomographie.

VI.3.1 Implémentation OpenCL : version finale

Nous avons donc présenté un total de 16 versions de notre algorithme de tomographie, dont trois NDRK et douze SWIK. De ces différentes versions, nous avons pu obtenir l'état final des optimisations de notre algorithme présenté au Tableau VI.14, qui correspond à

TABLE VI.13 – Optimisations fines : performances (Arria 10)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (s)	Performance relative (s)
T1_V6 : Pas de fusion des boucles (x, y, z), Équilibrage des tests	62	150	5.34	3.32
T1_V12 : Fusion des boucles (x, y, z)	62	150.57	5.35	3.32
T1_V13 : Pas d'équilibrage des tests	67	144.53	14.04	9.41

TABLE VI.14 – État final des optimisations de la rétroprojection (Arria10)

Tableau volume	Tableau sinogramme	Type de parallélisme	Tableaux α et β	Optimisations fines
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, non volatile	Pipeline x, y, z Déroulage φ x32	Déroulage	**Non**
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	SWIK	Locale	**Équilibrage**

l'implémentation *T1_V6*.

Nous avons comparé ces optimisations finales à nos optimisations hypothétiquement optimales présentées à la section VI.2, et le Tableau VI.15 présente leurs différences (les termes en gras dans les cases orangées correspondent aux optimisations différentes entre les deux versions).

TABLE VI.15 – Différences (en gras dans les cases oranges) entre la conjecture d'optimisation pertinente initiale (Tableau VI.2) et l'état final des optimisations (Tableau VI.14)

Tableau volume	Tableau sinogramme	Type de parallélisme	Tableaux α et β	Optimisations fines
Politique d'accès	***Politique d'accès***	***Boucles*** x, y, z, φ	***Boucle de copie***	***Fusion des boucles***
write_only, restrict	read_only, restrict, **non volatile**	Pipeline x, y, z Déroulage φ **x32**	Déroulage	Non
Type de mémoire	***Type de mémoire***	***Parallélisme général***	***Type de mémoire***	***Tests conditionnels***
Globale	Globale	SWIK	Locale	Équilibrage

En conclusion, l'analyse pertinente initiale de l'algorithme nous avait permis de dresser un état des optimisations hypothétiquement optimales (section VI.2), et nous pouvons constater que celles ci divergent peu par rapport aux optimisations effectivement optimales.

Nous avons pu constater un certain nombre de points de notre démarche d'optimisation sur cet algorithme de rétroprojection :

1. **Accès mémoire** : il faut veiller à adapter la zone mémoire au motif d'accès à la ressource (section VI.2.2).
2. **Parallélisme** : le parallélisme le plus fin est souvent le plus efficace (dans notre cas, le déroulage de la boucle la plus imbriquée) car c'est celle qui réplique le moins de ressources logiques périphériques (section VI.2.3).
3. **Dimensionnement mémoire** : pour des tableaux de taille raisonnable, leur allocation en mémoire locale est optimal (section VI.2.4). Toutefois, cette optimisation a peu d'impact sur les performances générales de l'algorithme, le goulot d'étranglement étant au niveau de l'accès au sinogramme. En effet, en terme de temps d'exécution mesuré, les trois optimisations (mémoire constante, locale, et registre à décalage) sont presque identiques.
4. **Optimisations fines** : l'équilibrage des accès mémoires et des branchements conditionnels (section VI.2.5) est important pour obtenir un pipeline efficace. Par contre, la fusion des boucles n'est pas toujours efficace, comme nous en avions fait la démonstration à la section IV.1.2.4.

Dans la section suivante, nous allons comparer la meilleure optimisation obtenue sur FPGA en OpenCL à différentes versions sur CPU et GPU.

VI.3.2 Comparaison de la consommation et des performances sur CPU - GPU - FPGA

Nous avons comparé le temps d'exécution mesuré sur notre Arria 10 (FPGA) au temps d'exécution mesuré sur la 1080 Ti (GPU bureautique), sur la Jetson TX2 (GPU embarqué), et le CPU d'Intel E5-2667, présentées au Tableau V.1.

Sur le FPGA, l'implémentation retenue est la *T1_V6*. Sur le CPU, nous avons implémenté une version à l'aide d'OpenMP pour tirer parti des 6 cœurs de l'architecture et, sur GPU, nous utilisons un code déjà optimisé tiré de la librairie *opgpuTomoGPI* développée au laboratoire L2S [Gac and Djafari, 2014].

Le Tableau VI.16 résume les différentes performances obtenues sur ces architectures.

Le temps d'exécution de l'algorithme sur le FPGA est certes 12 fois plus rapide que la version CPU multi-cœur, mais cela reste bien moins performant que les versions optimisées sur les deux GPUs. En conséquence, même si la puissance sur notre exécution du FPGA est la plus faible des quatre architectures, l'énergie consommée sur la durée de l'algorithme est également plus importante par rapport aux deux versions GPUs.

Dans notre cas de figure, la fréquence élevée, le grand nombre de cœurs des GPUs, et le parallélisme massif inhérent à l'algorithme rend ces plateformes particulièrement performantes par rapport aux FPGAs, qui peinent à rivaliser avec le débit de calcul des

TABLE VI.16 – Puissance et énergie consommée des optimisations les plus rapides de la rétroprojection sur CPU/GPU/FPGA

Architecture	Puisssance (W)	Temps d'exécution mesuré (s)	Énergie pour 256^3 voxels (mWh)
CPU Intel E5-2667	47	66	862
FPGA Intel Arria 10	9.9	5.340	14.65
GPU 1080 Ti	237	0.014	0.92
Jetson TX2	12.9	0.253	0.91

GPUs.

Toutefois, afin d'évaluer l'efficacité des "cœurs" élémentaires de nos optimisations, nous définissons le critère d'intérêt suivant :

Nombre de cycles nécessaires pour effectuer la mise à jour d'un voxel[1] par cœur ou pipeline élémentaire (noté η_{PE}) :

$$\eta_{PE} = \frac{TEM * F * \#PE}{\#MAJ_{voxels}} \tag{VI.3}$$

où $TEM =$ Temps d'exécution mesuré (s),
$F =$ Fréquence (Hz),
$\#PE =$ Nombre de cœurs/pipelines élémentaires,
$\#MAJ_{voxels} =$ Nombre de mises à jour de voxels

A partir de ce critère, nous comparons l'efficacité des différentes optimisations en s'appuyant sur les résultats du Tableau VI.17.

TABLE VI.17 – Comparaison de l'efficacité des différentes optimisations sur CPU, GPU et FPGA pour 256^4 mises à jours de voxel

Architecture	TEM (s)	PE ou cœurs physiques	Fréquence (MHz)	η_{PE}
CPU Intel E5-2667	66	6	2900	267.4
FPGA Intel Arria 10	5.340	32	150	5.97
GPU 1080 Ti	0.014	3584	1481	17.3
Jetson TX2	0.253	256	1300	19.6
FPGA VHDL*	–	–	–	1

* [Gac et al., 2008]

Cette fois-ci, on remarque que, avec nos spécifications haut niveau en OpenCL, l'outil

permet de traiter une mise à jour de voxel tous les 6 cycles pour chaque cœur élémentaire, ce qui est plus efficace que les architectures GPU et CPU. Par contre, si nous comparons l'efficacité de la meilleure optimisation OpenCL par rapport à l'implémentation VHDL sur FPGA [2] détaillé dans l'article [Gac et al., 2008], les outils ne permettent pas encore suffisamment de s'en rapprocher avec les leviers d'optimisation actuels, ce qui ouvre donc la voie à la recherche de nouveaux leviers d'optimisation.

VI.3.3 Conclusions

Dans ce chapitre, nous avons appliqué notre méthodologie d'accélération de code OpenCL sur FPGA, et nous pouvons constater que :

— Il est possible, en appliquant une démarche cohérente d'optimisation, d'obtenir une accélération significative d'un algorithme à partir d'une première version OpenCL fonctionnelle.

— La performance brute ainsi que l'énergie consommée par notre meilleure version OpenCL sur FPGA est meilleure que sur CPU multi-cœur, mais moins bonne que sur les GPUs présentés.

— Toutefois, le débit de calcul par pipeline élémentaire sur FPGA en OpenCL est plus efficace que sur les GPUs et le CPU, mais reste moins performant qu'une implémentation VHDL codée à la main.

Malgré tout, le compilateur OpenCL pour FPGA d'Intel reste un outil assez puissant, générant une implémentation à partir de notre code OpenCL certes moins efficace qu'un algorithme codé à la main en VHDL [Gac et al., 2008], mais en un temps de développement beaucoup plus court (4 mois pour OpenCL contre plus d'un an pour un implémentation avec un cache 3D de préchargement mémoire en VHDL).

Plus globalement, le manque de bus mémoires adaptés à une alimentation massive de données en parallèles sur un FPGA par rapport à un GPU reste un frein majeur pour certaines catégories d'algorithmes.

Ces résultats permettent toutefois d'envisager d'implémenter d'autres algorithmes de tomographie en OpenCL sur FPGA. En effet, certaines classes d'algorithmes, ont été délaissés car peu efficaces sur les GPUs, notamment en raison de branchements conditionnels, et une perspective intéressante serait d'évaluer l'efficacité de ces algorithmes sur FPGA en utilisant OpenCL.

2. Ces architectures dédiées en VHDL permettent d'obtenir une efficacité proche de l'optimal avec la mise à jour d'un voxel tous les cycles d'horloge.

Chapitre VII

Radar et systèmes d'écoute électromagnétique (Xilinx)

Sommaire

Ce chapitre est consacré à la présentation de l'accélération de deux cas d'études industriels développés au sein de Thales DMS France.

La motivation dans l'élaboration d'une démarche d'accélération des algorithmes est ici double. D'une part, il s'agissait d'évaluer la pertinence de la programmation des FPGAs en OpenCL comme alternative aux CPUs et au GPUs pour l'accélération des simulations, et d'autre part, il s'agissait d'étudier la convergence avec les outils OpenCL entre la description haut niveau d'un modèle d'étude et l'implémentation bas niveau d'un modèle contraint (VHDL/Verilog) utilisé en embarqué sur les systèmes réels.

VII.1 Vers une évolution régulière des différents niveaux de modélisation

Dans un projet industriel sur cible matérielle, il existe un certain nombre de niveaux de modélisation, illustrés à la Figure VII.1.

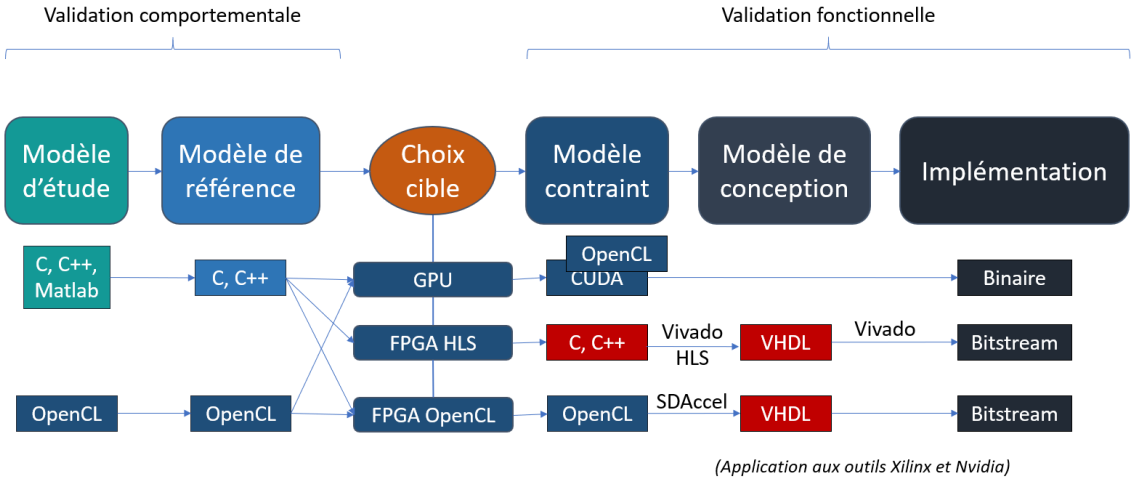


FIGURE VII.1 – Les différents niveaux de modélisation d'un algorithme

La syntaxe entre le modèle d'étude et le modèle de conception sur cible FPGA divergent souvent fortement. La promesse de Xilinx et d'Intel avec leurs solutions de programmation logicielle est justement de permettre une évolution régulière entre le modèle d'étude et le modèle contraint FPGA à l'aide des outils OpenCL. Du côté de Xilinx par exemple, les projets définis avec SDAccel en OpenCL génèrent un projet Vivado HLS qui appelle ensuite les outils Vivado. Il est donc possible d'affiner notre implémentation, au fur et à mesurer que l'on descend vers les couches bas niveau, avec les différents outils déjà utilisés dans le domaine du hardware.

Cette possibilité ouvre de nouvelles perspectives pour la définition de projets industriels. En effet, cette convergence entre toutes les couches, du modèle d'étude au modèle de conception embarqué peut être un avantage indéniable, non seulement vis-à-vis de la lisibilité et de l'évolution du code, mais aussi pour permettre aux différents métiers de mettre au point des optimisations communes.

VII.2 Simulateur d'environnements synthétiques : accélération d'un modèle de brouilleur radar

VII.2.1 Présentation du cas d'étude et enjeux

Le traitement radar est un domaine nécessitant une grande puissance de calcul. Dans le domaine aéroporté, tant civil que militaire, les radars ont une importance prépondérante, et, pour garantir la fiabilité du système global, il est nécessaire de le tester en amont. Une étape essentielle de validation des éléments du système consiste à les simuler aussi précisément que possible, afin de les coupler à un générateur de scénarios. La simulation complète permet alors de tester en toute robustesse les algorithmes élémentaires ainsi que la chaîne de traitement des systèmes.

Aussi, construire un jumeau numérique du système final peut s'avérer être un réel avantage pour tester la complexité de chaque sous-système, et pour, dans certains cas, affiner le choix des cibles matérielles du système réel.

Cette représentation virtuelle du système présente deux avantages majeurs, en premier lieu, associer ce clone virtuel à un générateur de scénarios permet de le tester de manière exhaustive, améliorant ainsi la fiabilité des traitements, en second lieu, cette approche génère une économie considérable si l'on compare son coût à celui d'essais en vol.

Dans ce cas d'étude, nous nous sommes intéressés à un outil développé par Thales DMS France, qui permet de simuler des environnements, en générant les échos numériques correspondants qui alimentent des modèles radars pour en valider les traitements. Ce simulateur inclut de nombreux environnements, comme des nuages, des autoroutes, différents sols, ou encore des cibles mouvantes. Il est utilisé pour tester en amont à la fois les traitements, mais aussi la réponse à certains environnements du radar, tout en validant dans une moindre mesure certains capteurs analogiques.

La complexité croissante des systèmes radars réels créent un besoin au niveau de l'accélération des simulations, et c'est pourquoi nous avons intégré ce cas d'étude dans nos travaux d'évaluation de l'approche OpenCL pour la programmation des FPGAs.

Nous présentons dans la section VII.2.1, le contexte d'étude, avec l'introduction du concept de radar, la présentation de l'outil de simulation et le modèle de brouillage aéroporté dans un environnement simulé. Puis, dans la section VII.2.1.4, nous analysons les spécificités de l'algorithme choisi et, dans la section VII.2.2, discutons de notre démarche d'optimisation en OpenCL sur le FPGA Arria10 d'Intel, décrit dans le Tableau V.1. La section VII.2.3.1 est consacrée à la présentation rapide de nos implémentations sur GPU, inspiré de nos travaux précédents [Martelli et al., 2019a], puis nous discutons à la section des résultats obtenus tant en termes de performance brute qu'en comparaison avec les autres architectures, avant de dresser les premières conclusions quant à l'efficacité des outils OpenCL, à la section VII.2.3.4.

VII.2.1.1 Radar : concepts préliminaires

Les avancées de la recherche dans le domaine des radiocommunications et les besoins stratégiques ont conduit au développement des systèmes de détection et télémétrie par radio électricité, communément désignés par le terme de RADAR. Utilisé dans

un vaste panel d'applications, allant de la météorologie aux systèmes de défense, c'est aujourd'hui une technologie indispensable. Le principe physique général d'un radar est qu'après émission d'une impulsion électromagnétique dans une direction donnée, si le signal rencontre une cible, cette impulsion se réfléchit, et une partie d'entre elle retourne vers le radar. En mesurant le délai temporel entre émission et réception de l'écho, il est alors possible de corréler, à l'aide de (VII.1), la distance entre le radar et la cible.

$$R = \frac{c * T_{ER}}{2} \tag{VII.1}$$

où $R =$ distance radar-cible,
$c =$ vitesse de la lumière dans l'air,
$T_{ER} =$ délai temporel entre émission et réception.

Le radar Doppler, classe particulière de radars, utilise l'effet Doppler-Fizeau. Le signal émis par l'antenne a une fréquence précise F_e, et l'écho reçu de la réflexion sur la cible a une autre fréquence F_r. A partir de l'effet Doppler, on considère en première approximation l'équation (VII.2) qui permet de calculer la vitesse radiale relative v de l'écho considéré (positive en rapprochement relatif).

$$F_r = F_e + \frac{2v}{\lambda} \tag{VII.2}$$

où $F_e =$ fréquence d'émission,
$F_r =$ fréquence de réception,
$v =$ vitesse radiale relative de l'écho considéré,
$\lambda =$ longueur d'onde du signal émis.

Cela permet de construire une carte Distance Ambiguë Vitesse Ambiguë (DAVA) comme illustré à la Figure VII.2, dont l'objectif est d'afficher de manière synthétique les cibles détectées tout en les caractérisant en distance et en fréquence Doppler.

Chaque carré de la carte représente une cible détectée par les premiers traitements du radar. Une carte DAVA réelle contient des échos additionnels en provenance de l'environnement radar.

VII.2.1.2 Simulateur d'Environnements Numériques

Le Simulateur d'Environnements Numériques (SEN) est un outil industriel de simulation et de validation, en support du développement d'algorithmes innovants de traitement du signal pour les radars aéroportés. Avec celui-ci, les radars sont précisément modélisés afin de simuler de la manière la plus représentative possible les échos réfléchis par un environnement qui peut inclure nuages, forêts, déserts, océans, cibles aériennes, marines, ou terrestres, mais également, entre autres, des brouilleurs, comme illustré à la Figure VII.3.

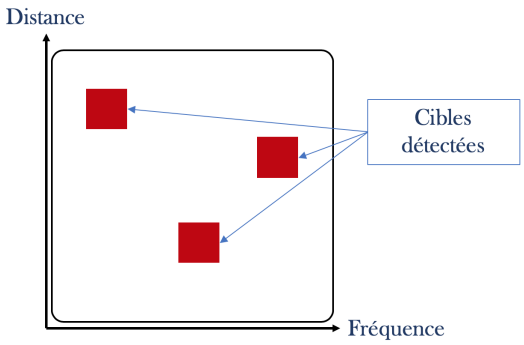


FIGURE VII.2 – Carte Distance Ambiguë Vitesse Ambiguë

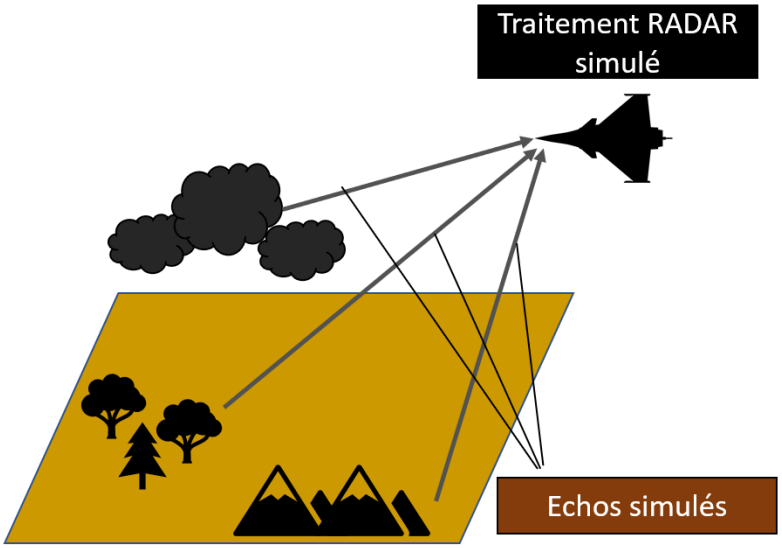


FIGURE VII.3 – Simulateur d'Environnements Numériques

Les échos générés par le SEN sont localisés à la sortie complexe démodulée du radar, tout en prenant en compte certains filtres et composants analogiques. Selon le type et la représentativité demandée des environnements numériques développés, les temps de génération des échos sont plus ou moins longs. Lors du couplage de ce moyen de stimulis avec l'implémentation matérielle des algorithmes et traitements sous test, un temps de génération des stimuli quasi temps réel est requis.

Ainsi, nous avons choisi d'accélérer, parmi la bibliothèque des environnements disponibles, le Brouilleur Corrélé Localisé en Distance et en Fréquence (BCLDF), dont la génération nécessite une accélération pour tenir ce temps réel.

VII.2.1.3 Modèle de brouillage aéroporté dans un environnement simulé

Comme expliqué dans la brève introduction sur les traitements radars, l'une des étapes de notre algorithme de traitement radar est de construire une carte DAVA, qui illustre et caractérise en distance et en vitesse les cibles détectées par le radar. Le but du

Brouilleur Corrélé Localisé en Distance et en Fréquence (BCLDF) est de brouiller le radar afin de camoufler la position d'une cible, en saturant une zone localisée autour de celle ci sur la carte.

La Figure VII.4 illustre ce principe, et montre quelle incidence ce modèle de brouillage aurait sur une carte DAVA (comparaison avec Fig. VII.2).

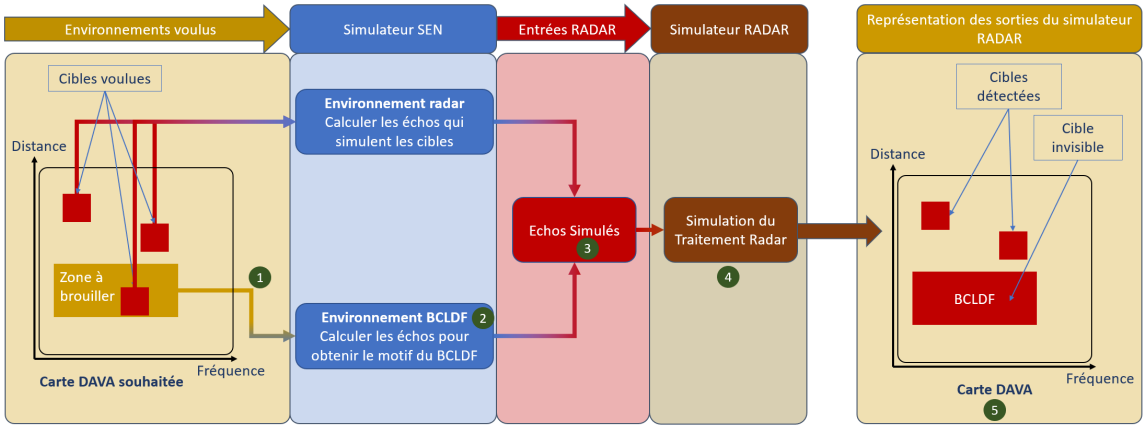


FIGURE VII.4 – Modèle de brouillage et simulateur d'environnements numériques

Pour obtenir le brouillage d'une zone, son étendue en distance et en fréquence sur la carte DAVA souhaitée est prise en paramètre (*N°1*), puis, en inversant les traitements radar (*N°2*), nous obtenons les échos sous forme de signaux complexes correspondant aux cases brouillées (*N°3*).

En alimentant la simulation du traitement radar avec ces échos simulés (*N°4*), nous obtenons une carte DAVA (*N°5*) avec un brouillage localisé comme illustré à la Figure VII.4. L'effet d'un BCLDF académique sur un radar simulé est illustré à la Figure VII.5.

Nous avions comme version de référence de l'algorithme une implémentation sur CPU, intégrée au simulateur SEN, et notre objectif était d'implémenter des optimisations sur FPGA et GPU.

VII.2.1.4 Analyse de l'algorithme et protocole de test

Un BCLDF brouille donc une zone rectangulaire dont le nombre de cases en distance et en vitesse sont paramétrables. Le principe est d'appliquer un bruit sur cette zone, et le signal temporel de brouillage est obtenu à l'aide d'une compression d'impulsion inverse, suivi d'une transformée de Fourier rapide (*Fast Fourier Transform*) (FFT) inverse.

Plus précisément, une fois que le nombre de cases en distance et en fréquence du brouillage est choisi, pour calculer les signaux de brouillages correspondant aux échos que recevraient un radar, il faut :

— effectuer la convolution d'une réplique de la matrice DAVA, ce qui permet de passer en temporel sur l'axe des cases distance,

— effectuer une FFT inverse, qui permet de passer en temporel sur l'axe des fréquences, et appliquer en même temps les erreurs de phases des voies et des antennes ainsi qu'un gain paramétrique.

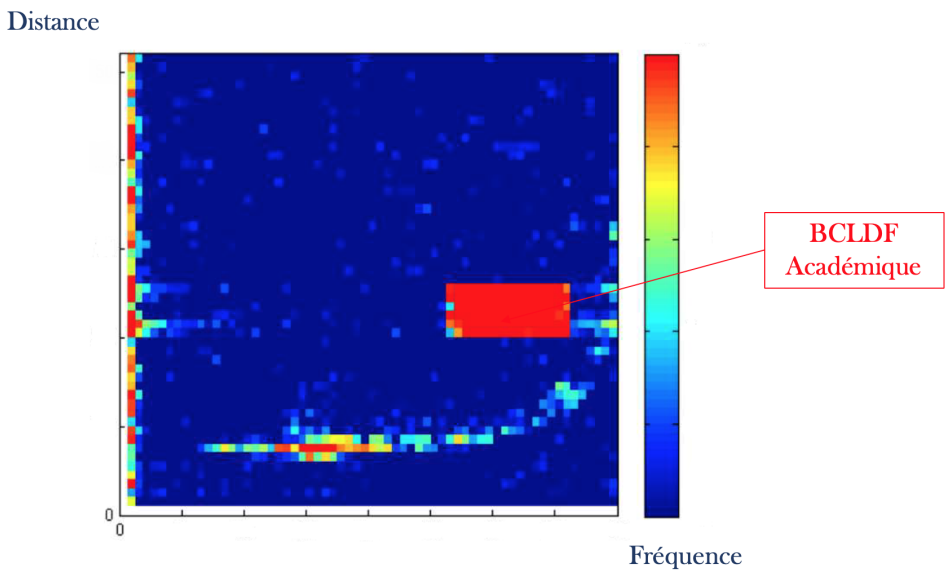


FIGURE VII.5 – Conséquences d'un BCLDF synthétique sur une carte DAVA (échelle supprimée volontairement).

L'algorithme repose donc sur une transformée de Fourier inverse ainsi qu'une convolution, toutes deux opérant sur des tableaux ou matrices de grandes dimensions.

VII.2.2 Exploration des optimisations sur FPGA

Il n'existe pas actuellement, à notre connaissance, de librairie FFT optimisée pour les FPGAs, et, les rares exemples d'implémentation en OpenCL ne sont pas adaptées aux dimensions de notre problème.

Aussi, la première étape consiste à traduire le code C en OpenCL, et à implémenter une version adaptée à notre problème d'un algorithme de FFT inverse.

Notre démarche d'optimisation FPGA, associée à différentes versions en OpenCL, consiste donc à :

— adapter le code C en OpenCL,

— implémenter une version OpenCL parallélisable de la FFT (*V1*),

— optimiser la localisation mémoire de la carte DAVA (*V1,V2,V3,V4*),

— paralléliser notre algorithme (*V5,V6,V7,V8*).

VII.2.2.1 Implémentation de la FFT (*V1*)

La version initiale sur CPU utilise un algorithme de FFT séquentiel. Aussi, la première étape a consisté à :

— convertir les structures de données complexes de C (pointeurs de pointeurs) en structures de données OpenCL (pointeurs simples),

— aligner les structures mémoires,

— implémenter une version parallélisable d'un algorithme de FFT inverse.

Dans notre cas, pour le calcul d'une FFT d'ordre $n = 4^k$, l'algorithme que nous avons choisi d'implémenter effectue une FFT sur 16 points, dont le principe, détaillé dans l'article [Sivakumar M et al., 2015], permet de décomposer un signal à N points dans le domaine temporel, en N signaux composés chacun d'un seul point dans le domaine temporel également. La Figure VII.6 illustre le principe de cet algorithme.

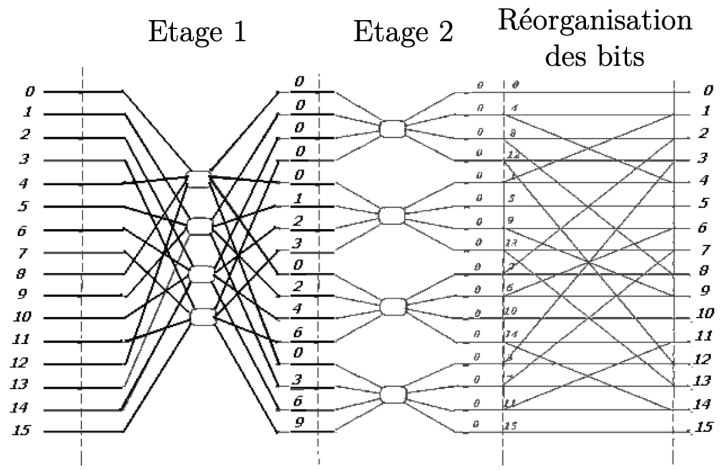


FIGURE VII.6 – Calcul d'une FFT 16 points à l'aide de radix-4.

Source: Article [Sivakumar M et al., 2015]

En utilisant une décomposition en base 4 (*radix-4*), les étages de calculs nécessaires sont réduits, et, pour calculer une FFT à 16 points, il n'y a besoin que de $log_4(16) = 2$ étages. Si nous avions choisi une FFT à base de *radix-2*, il aurait fallu $log_2(16) = 4$ étages de calcul pour un même algorithme.

Comme nous ne pouvons garantir que la taille de notre FFT soit exactement une puissance de 4, il suffit de prendre comme ordre la puissance de 4 supérieure à la taille réelle de la FFT à calculer.

En plus de l'implémentation de la FFT, nous avons choisi la mémoire globale avec cache comme localisation mémoire pour la matrice DAVA. La pertinence de ce choix est discuté à la section suivante.

Résumé des optimisations (*V1*) : FFT à 16 points, pipeline des boucles, carte DAVA en mémoire globale avec cache.

VII.2.2.2 Choix de la localisation mémoire

VII.2.2.2.a Mémoire d'interface (*V2, V3*)

L'optimisation de la localisation en mémoire de la matrice représentant la carte DAVA, est indispensable car la matrice est trop volumineuse pour être implémentée en mémoire locale uniquement. Les trois choix possibles pour la stocker sont donc :

— mémoire globale avec cache (*V1*),

— mémoire constante (*V2*),

— mémoire globale sans cache (*V3*).

Les optimisations correspondantes sont présentées dans le Tableau VII.1.

TABLE VII.1 – Choix de la localisation mémoire DAVA : performances (Xilinx KCU115)

Version	Utilisation ressources empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (ms)	Performance relative (ms)
V1 : mémoire globale avec cache	21	270	1422	299
V2 : mémoire constante	21	272	1045	219
V3 : mémoire globale sans cache	12	287	1961	235

Notre démarche d'optimisation consiste à minimiser le temps d'exécution de nos implémentations. Ainsi, même si la *V3* utilise moins de ressources que la *V2*, cette dernière reste plus efficace par rapport à son utilisation logique (performance relative).

Nous avons donc choisi de stocker la carte DAVA dans la mémoire constante (*V2*) ce qui est cohérent avec l'accès très fortement régulier de nos algorithmes (FFT et convolution).

VII.2.2.2.b Mise en cache locale manuelle (*V4*)

A partir de l'implémentation *V2*, nous avons évalué la pertinence d'implémenter une mise en cache supplémentaire. En effet, notre algorithme de FFT utilise des groupes de 16 valeurs contiguës, qui, une fois chargées, sont accédées dans un ordre non contigu, et il paraissait cohérent de les mettre en cache en mémoire locale. En effet, la particularité de ce type de mémoire sur FPGA est de garantir une latence d'accès fixe quelque soit l'élément auquel nous devons accéder.

Nous avons donc ajouté une nouvelle implémentation, mémoire constante avec mise en cache locale (*V4*), obtenue à partir de la version *V2*, dont les performances sont présentées au Tableau VII.2.

TABLE VII.2 – Implémentation manuelle d'un cache local : performances (Xilinx KCU115)

Version	Utilisation ressources empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (ms)	Performance relative (ms)
V2 : mémoire constante	21	272	1045	219
V4 : mémoire constante avec cache local	26	262	627	163

Nous avions vu que la mémoire constante était la plus pertinente au vu de l'accès aux données dans notre algorithme, et, pour une empreinte mémoire légèrement plus importante, l'implémentation d'une mise en cache locale en plus du cache de mémoire constante permet d'augmenter encore un peu les performances de notre implémentation.

Résumé des optimisations (*V4*) : FFT à 16 points, pipeline des boucles, carte DAVA en mémoire constante avec cache local implémenté manuellement.

VII.2.2.3 Choix du parallélisme (*V5-8*)

Pour exprimer le parallélisme de notre algorithme, nous avons deux options :

— la parallélisation des boucles qui traitent chaque étage de la FFT (chaque FFT à 16 points),

— la réécriture de la convolution en dataflow (flot de données) [Shi et al., 2019], au lieu d'un parallélisme de données NDRK.

Afin d'évaluer la pertinence de ces deux approches, nous avons procédé de manière incrémentale, et les différents types d'optimisations implémentés sont les suivants :

— FFT 16 points : déroulage x16, convolution NDRK : *V5*

— FFT 16 points : déroulage x32, convolution NDRK : *V6*

— FFT 16 points : déroulage x16, convolution dataflow : *V7*

— FFT 16 points : déroulage x24, convolution dataflow : *V8*

Le tableau VII.3 résume les résultats obtenus pour ces différentes optimisations de parallélisme, et nous y avons rajouté la meilleure optimisation actuelle (*V4*) pour avoir une référence de comparaison.

TABLE VII.3 – Types de parallélisme (BCLDF) : performances (Xilinx KCU115)

Version	Utilisation empirique (%)	Fréquence (MHz)	Temps d'exécution mesuré (ms)	Performance relative (ms)
V4 : mémoire constante avec cache local,	26	262	627.8	163
Expression du parallélisme				
V5 : Déroulage FFT x16, convolution NDRK	54	221	57.6	31
V6 : Déroulage FFT x32, convolution NDRK	67	209	33.1	22
V7 : Déroulage FFT x16, convolution dataflow	58	246	41.4	24
V8 : Déroulage FFT x24, convolution dataflow	86	233	23.1	20

Tout d'abord, la réécriture de notre convolution en dataflow utilise plus de ressources que la convolution en parallélisme de données. Le facteur maximal de déroulage de la FFT avec convolution en dataflow en est réduit (x24 au lieu de x32).

Toutefois, bien que consommant le plus de ressources, la version *V8 (Déroulage FFT x24, convolution dataflow)* est la plus rapide des 5 versions présentées.

VII.2.2.4 Conclusion des implémentations FPGA

Nous avons pu obtenir, à partir de notre première version de référence (*V1*), un temps d'exécution de 23 ms contre 1422 ms initialement, soit un facteur d'accélération de 62. Pour cette version (*V8*), nous avons dû réécrire la quasi totalité du code, les modifications l'impactant en profondeur. En effet, les algorithmes de FFT et de convolution ont été adaptés à une exécution plus efficace sur FPGA, et les structures de données ont elles aussi été modifiées pour satisfaire l'alignement sur notre plateforme.

Si la contrainte prioritaire est le temps de développement, la version *V6* est un bon compromis. En effet, il n'y a pas besoin de réécrire l'algorithme de convolution et les performances obtenues sont correctes (33 ms au lieu de 1422 ms pour la version *V1*, soit un facteur d'accélération de 43).

En conclusion, et si conceptuellement le principe de l'algorithme reste inchangé, la version la plus optimisée sur FPGA en OpenCL diverge amplement de la version de référence sur CPU.

VII.2.3 Bilan : CPU, GPU, FPGA

VII.2.3.1 Implémentations GPU

Afin de comparer les différentes architectures, nous avons donc choisi de traduire l'algorithme de brouillage initial sur CPU en différentes implémentations sur GPU en utilisant deux langages différents :

— OpenACC, qui permet, à l'aide de directives préprocesseurs sans modifications en profondeur du code, d'exprimer le parallélisme d'un algorithme ainsi que les copies mémoires entre CPU et GPU,

— CUDA, qui est le langage de prédilection pour obtenir les meilleures performances sur les GPUs Nvidia.

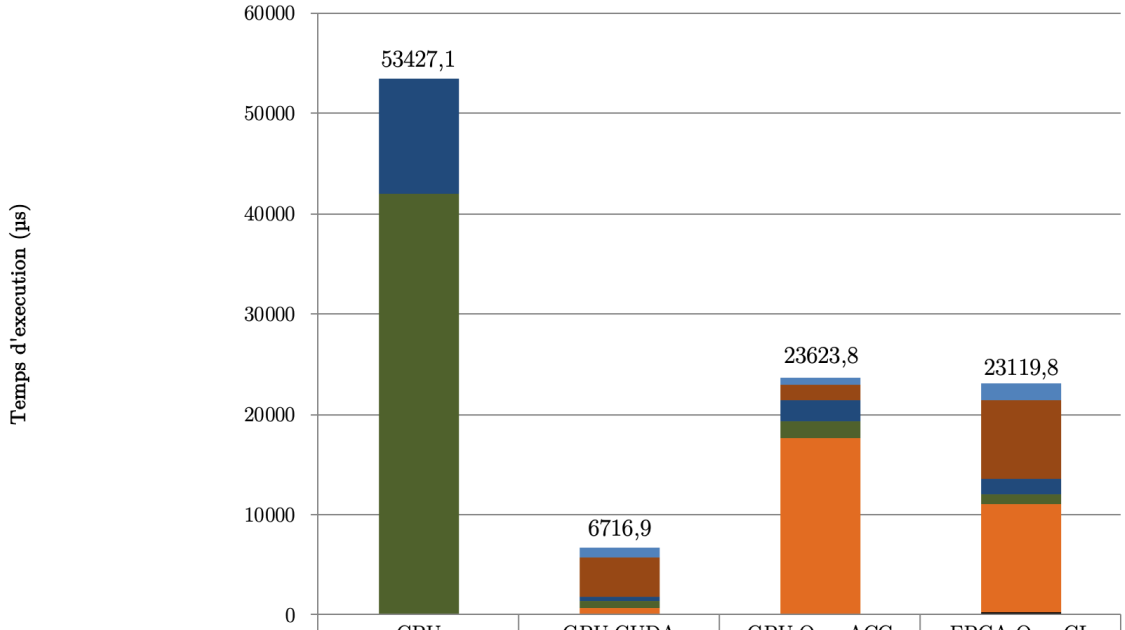
L'intérêt était d'évaluer le compromis entre le temps de développement sur GPU et l'accélération du temps d'exécution d'une application réelle.

Comme l'algorithme de référence qui inclut une FFT non parallélisable, nous avons choisi d'utiliser la version optimisée cuFFT de Nvidia. Il est donc nécessaire ici encore de réécrire une première version fonctionnelle de l'algorithme, et nous avons repris les transformations effectuées lors de la conversion du code C en OpenCL, notamment la réécriture des pointeurs de pointeurs en pointeurs simples, et l'alignement des structures.

VII.2.3.2 Comparaison détaillée des temps d'exécution

Afin d'analyser plus en profondeur la différence dans l'exécution sur les différentes architectures, nous avons segmenté cet algorithme en plusieurs étapes.

La comparaison du temps d'exécution mesuré est présentée à la Figure VII.7 pour les versions suivantes : CPU de référence, GPU OpenACC, GPU CUDA, FPGA OpenCL la plus optimisée (*V8*).



	CPU	GPU CUDA	GPU OpenACC	FPGA OpenCL
Liberation GPU ou FPGA	0	934,6	624,7	1701,6
Transfert vers CPU ou FPGA	0	4021	1577,8	7841,3
FFT inverse	11498,6	372,1	2129,2	1527,1
Décompression d'impulsion	41928,5	688,2	1633,5	954,2
Transfert vers GPU ou FPGA	0	698,3	17653	10749,9
Initialisation GPU ou FPGA	0	2,7	5,6	345,7

FIGURE VII.7 – BCLDF : Temps d'exécution total et détails par fonctions (CPU, GPU OpenACC, GPU CUDA, FPGA OpenCL(V8)).

De cette figure, nous pouvons dresser un certain nombre d'observations :

— Versions FPGA et GPU : elles sont toutes plus rapides que la version CPU de référence, avec des facteurs d'accélération de 2.31, 2,26, et 7,95 pour respectivement les versions FPGA OpenCL, GPU OpenACC, et GPU CUDA. En ne considérant que les calculs (en omettant les transferts mémoires), les facteurs d'accélération sont respectivement de 21.53, 14.20, et 50.39.

— Versions GPU : les étapes de calculs (décompression et FFT inverse) prennent un temps négligeable par rapport aux temps de transferts entre CPU et GPU.

— Version GPU OpenACC : elle est peu efficace pour transférer des données du CPU au GPU par rapport à son équivalente CUDA.

— Version FPGA : les étapes de calculs sont plus rapides que la version GPU OpenACC, mais cette implémentation est pénalisée par les transferts mémoires CPU/FPGA.

Toutes les versions implémentées semblent donc pertinentes par rapport à la version CPU de référence, mais nous discutons de l'efficacité énergétique de ces différentes implémentations à la section suivante.

VII.2.3.3 Efficacité énergétique

Nous avons donc, pour notre modèle de brouillage, implémenté plusieurs versions en OpenCL sur FPGA, ainsi qu'en OpenACC et CUDA sur GPU. Nous résumons les meilleures optimisations obtenues par catégories, dans le Tableau VII.4, dont les temps d'exécutions détaillés ont été présentés à la Figure VII.7.

TABLE VII.4 – Puissance et énergie consommée des meilleures optimisations du modèle de brouillage sur CPU/GPU/FPGA

Architecture	Puissance (W)	Temps d'exécution mesuré (μs)	Énergie (μWs)
CPU - Intel E5-2667	39	53.4	2082.6
FPGA - Xilinx KCU115	14.5	23.1	335.0
GPU - 1080 Ti (OpenACC)	218	23.6	5144.8
GPU - 1080 Ti (CUDA)	216	6.72	1451.5

Bien que toutes nos implémentations soient toutes plus rapides que la version de référence CPU, nous constatons que la version OpenACC est sous-optimale, en ce qu'elle consomme presque 16 fois plus d'énergie que la version FPGA pour un temps d'exécution légèrement plus long.

L'implémentation la plus optimale sur notre carte Xilinx (*V8*) quant à elle est la plus efficace en terme de performance par watt, mais la version sur la carte graphique 1080Ti codée en CUDA reste la plus performante, avec une consommation énergétique certes plus importante que la version FPGA, mais réduite par rapport à la version de référence.

VII.2.3.4 Conclusion sur la démarche d'optimisation

A partir d'un code CPU de référence, nous avons dû, tant pour nos versions FPGAs que GPUs, réécrire le code de manière substantielle, en raison d'une inadéquation majeure de la version initiale avec les différents modèles de parallélisme.

Finalement, le choix final de la meilleure optimisation dépend des contraintes voulues dans le contexte de l'application.

— Si la performance par watt est la contrainte forte, alors l'implémentation *V8* en OpenCL sur FPGA est la plus efficace énergétiquement, tout en étant plus rapide (x2.31) que la version de référence CPU.

— Si par contre, la performance brute est la contrainte prioritaire, alors la version CUDA sur GPU est la plus optimale, car elle permet une accélération d'un facteur 7.95 par rapport à la version de référence CPU.

VII.3 Implémentation d'un modèle de référence pour la génération de signaux numériques superhétérodynes

VII.3.1 Présentation du cas d'étude et enjeux

VII.3.1.1 Synoptique du projet

Le générateur de signal numérique superhétérodyne est capable de générer des échantillons numériques en temps réel, qui peuvent être interfacés pour permettre la validation des traitements concernant l'espionnage des paramètres de configuration d'écoute électromagnétique (bande de fréquence, gain, antennes sélectionnées, etc.). Il est composé de différents blocs d'algorithmes, détaillés à la Figure VII.8.

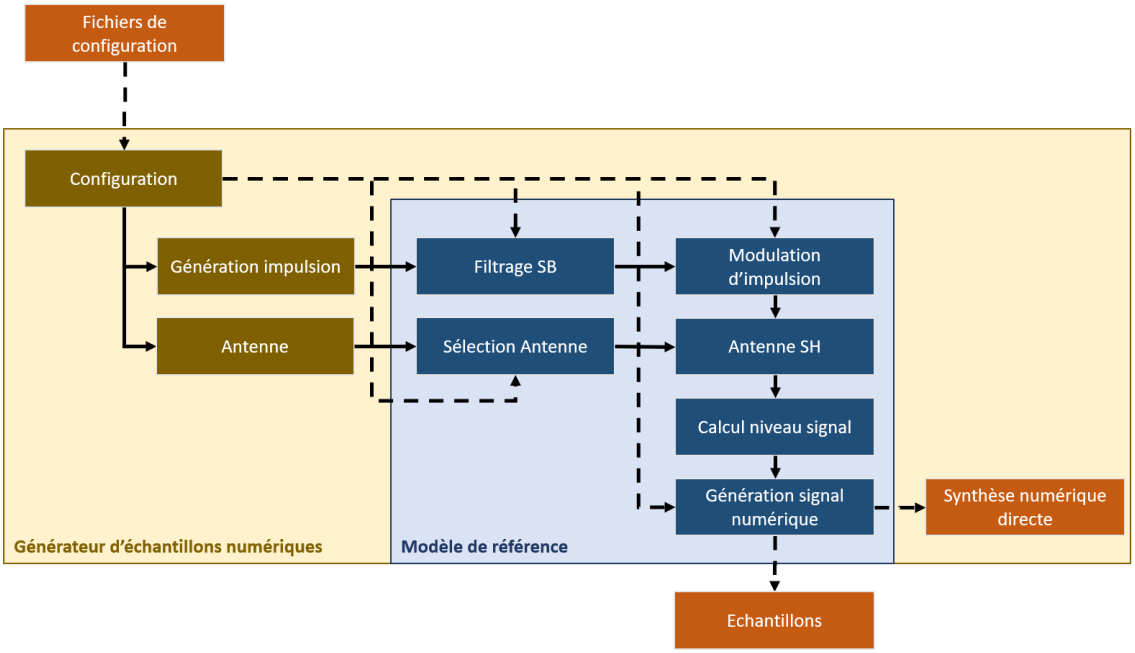


FIGURE VII.8 – Synoptique du générateur de signal numérique

A partir des différents fichiers de configuration, qui décrivent notamment la position et les caractéristiques des émetteurs, l'étape de **Configuration** permet de calculer les paramètres comme le bruit des impulsions, le choix du panneau de l'antenne, l'interférométrie, la fréquence de la bande d'écoute, ou encore la modulation intrapulse en fonction des paramètres de simulation voulus. Ensuite, la **génération d'impulsion** va lire le scénario, calculer le temps de trajet de chaque impulsion entre les émetteurs et les antennes de réception, et générer les ondes entretenues [1] ainsi que les impulsions qui sont censées être reçues par l'antenne réelle.

En parallèle, le bloc **Antenne** calcule pour chaque émetteur le gain et les phases pour l'ensemble des configurations antennaires à l'aide d'un modèle paramétrique. Les traitements suivants constituent le modèle de référence de notre projet. Le **Filtrage SB** teste

1. Onde électromagnétique continue d'amplitude et de fréquence constantes.

si les impulsions produites par la génération d'impulsion sont valides et si elles appartiennent à la bande d'écoute. L'étape de **Modulation d'impulsion** consiste à faire évoluer les impulsions courantes, ce qui passe par en créer de nouvelles suivant la sortie du **Filtrage SB**, par mettre à jour celles existantes, et par supprimer celles qui sont terminées.

Pour chacune des impulsions précédentes, l'**Antenne SH** récupère les paramètres d'antennes et les applique aux impulsions des différents émetteurs, en calculant le gain et la phase suivant ces paramètres. Ensuite, l'étape **Calcul niveau signal** consiste, à partir des impulsions et des paramètres d'antennes, à calculer les niveaux de bruits correspondants, et à les envoyer au dernier bloc : la **Génération du signal numérique** à proprement parler.

Cette partie va, pour toutes les impulsions présélectionnées, calculer leurs fréquences et enregistrer chaque échantillon dans un fichier de sortie.

Comme illustré à la Figure VII.8, dans le cas d'un système réel, ces échantillons sont envoyés à un bloc sur le FPGA qui s'occupe de la synthèse numérique directe (DDS - Direct Digital Synthesis).

Cette étape permet de produire une onde analogique par la manipulation numérique d'un signal d'horloge afin d'obtenir une résolution fine en fréquence sur une plage étendue.

L'intérêt d'implémenter le modèle de référence sur FPGA est donc d'autant plus intéressant que les traitements dans le système réel sont censés s'interfacer directement avec d'autres traitements sur FPGA.

VII.3.1.2 Analyse de l'algorithme et protocole de test

Nous avons choisi, pour nos simulations, un cadre de scénario représentatif du cas de fonctionnement réel, dont les paramètres sont :

— Temps simulé du scénario de 30 secondes,

— 32 émetteurs maximum,

— 8 impulsions simultanés maximum.

— 4 voies maximum

Aussi, il est possible d'alimenter notre simulation avec des scénarios différents, si tant est qu'ils respectent les caractéristiques précédentes. Nous avons choisi trois scénarios, dont les caractéristiques sont présentées au Tableau VII.5

TABLE VII.5 – Génération d'échantillons : scénarios de tests

Scénario	Nombre d'émetteurs	Nombre de voies
N°1	3	4
N°2	10	4
N°3	32	4

Entre chacun des blocs présentés à la sous-section précédente, il y a un certain nombre de données partagées, comme par exemple les paramètres d'antenne, le tableau des impulsions, et une attention particulière doit être consacrée à l'analyse pertinente du partage de ces ressources ainsi qu'à leur dimensionnement.

La plupart des blocs contiennent trois types de boucles, portant sur les différents émetteurs, voies, et impulsions. Dans certains cas, ces boucles sont imbriquées, mais, quelque soit la configuration, il n'y a pas de dépendance de données entre chaque itération successive. Si la parallélisation de données semble bien se prêter à ce cas d'étude, le nombre d'itérations à paralléliser est au maximum de $32 * 8 * 4 = 1024$ dans le cas où trois boucles sont imbriquées.

Le modèle de référence sur CPU est codé en C++ et la structure des objets passe par l'utilisation de pointeurs et de copies locales. Une option possible pour optimiser les performances est de réduire ces copies afin d'alléger l'empreinte mémoire de notre algorithme et la redondance des données.

VII.3.2 Exploration des optimisations sur FPGA

Nous présentons, implémentées sur notre FPGA Xilinx KCU115, trois grandes étapes d'optimisations correspondant :

— à la première implémentation fonctionnelle en OpenCL : *V1*,

— à l'expression du parallélisme via le déroulage des boucles : *V2*,

— aux optimisations fines : *V3*.

Les résultats de ces optimisations seront présentés à la section VII.3.3.

VII.3.2.1 Implémentation OpenCL : version initiale (*V1*)

Cette première implémentation, qui sert par la suite de référence pour les autres optimisations OpenCL n'est que la traduction du modèle de référence CPU en OpenCL ciblant notre FPGA Xilinx. Dans ce cas d'étude, il n'y a pas d'adaptation particulière des types à effectuer, puisque ceux-ci sont déjà correctement dimensionnés. Également, nous laissons pour cette version toutes les boucles en pipeline.

La transformation majeure (en temps de développement) se situe au niveau de la réécriture du code. En effet, le code de référence en C++ utilise la programmation objet, qui s'appuie largement sur les accesseurs (get) et les mutateurs (set) [2] pour accéder et modifier les champs d'un objet. Or, sur FPGA, un appel de fonction est coûteux, et il faut donc restructurer le code pour accéder directement aux variables voulues sans passer par une représentation objet.

En analysant les variables accédées en lecture seule et régulièrement utilisées, nous pouvons calculer leur empreinte mémoire (à partir de leurs types) et choisir, si la place le permet, de les implémenter en mémoire locale ou constante.

Sur le FPGA Xilinx, nous avons pu stocker dans toutes nos implémentations ces variables de configuration dans la mémoire locale, permettant un accès optimal à celles-ci.

Résumé des optimisations courantes (*V1*) : transformation du code C++ en OpenCL , mise en mémoire locale des variables en lecture seule utilisées fréquemment, pipeline des boucles.

2. Fonctions d'accès et de modification d'objets en C++.

VII.3.2.2 Expression du parallélisme - déroulage des boucles (*V2*)

Nous avons pu voir pendant les cas d'étude précédents que le parallélisme le plus efficace sur les FPGAs en OpenCL reste le déroulage des boucles.

La seconde optimisation était donc logiquement d'essayer de paralléliser toutes les boucles des différents blocs. Mais, cette optimisation étant trop gourmande en ressources logiques, nous ne pouvions pas l'implémenter sur notre carte Xilinx.

Aussi, nous nous sommes concentrés en priorité sur les boucles les plus imbriquées afin d'obtenir un grain fin de parallélisme.

Cette implémentation, comme présentée au Tableau VII.6, est plus efficace que la version initiale, mais en analysant le chemin des données sur le FPGA, nous avons pu observer une sous-utilisation des ressources suivant les scénarios d'utilisation.

En effet, les boucles, dans le modèle de référence étaient effectuées non pas sur le nombre d'éléments réels mais sur le nombre maximum d'éléments par catégorie, et notre traduction en OpenCL avec déroulage de boucle nous donne un code similaire à celui présenté à l'Algorithme 15.

Algorithme 15 : Génération d'échantillons - exemple de boucles

```
1 #pragma unroll EMETTEURS_MAX
2 for ( e = 0 to EMETTEURS_MAX - 1 ) {
3     emetteur = Tab_emetteur[i].NumEmetteur;
4     if emetteur != 0 then
          /* Calculs                                          */
5         #pragma unroll IMPULSIONS_MAX
6         for ( i = 0 to IMPULSIONS_MAX - 1 ) {
              /* Calculs                                      */
```

Dans le cas où le scénario n'a que quelques émetteurs par rapport aux 32 émetteurs maximums imposés par notre configuration, le déroulage de boucle sur l'ensemble de celle-ci entraînera une réplication excessive des ressources eu égard à leur utilisation réelle.

C'est pourquoi nous avons choisi de restructurer entièrement notre code, comme présenté à la sous-section suivante, pour mieux l'adapter aux différents scénarios.

Résumé des optimisations courantes (*V2*) : transformation du code C++ en OpenCL , mise en mémoire locale des variables en lecture seule utilisées fréquemment, déroulage des boucles.

VII.3.2.3 Équilibrage des tests conditionnels et communication inter-kernels (*V3*)

Chaque bloc de calcul réutilise en entrée des données de sortie du bloc précédent, en plus de certaines données qui sont accédées par tous en lecture seule, comme les paramètres de configuration.

Aussi, nous avons choisi d'implémenter un algorithme en dataflow, sur le principe du producteur/consommateur, où un kernel va envoyer dans une file d'attente les impulsions générés au fur et à mesure, et celles-ci seront consommées par le kernel suivant. Par

exemple, là où la génération d'impulsion génère un tableau d'impulsions qui est ensuite passé comme argument à la fonction de *Filtrage sous bande d'écoute* pour trier les impulsions valides, nous transformons les deux blocs pour que la génération d'impulsion envoie chaque impulsion une fois générée, et le bloc de filtrage catégorise ce dernier sans en attendre d'autres, avant de passer les impulsions triées au bloc suivant.

Cette optimisation, détaillée à la section IV.3.3, permet non seulement de fluidifier notre implémentation mais, par la localisation de ces files d'attentes dans les blocs RAM des FPGAs, de réduire la latence et d'accélérer la vitesse d'accès à ces structures, sans devoir repasser par la mémoire globale du FPGA.

Ainsi, quelque soit le nombre d'émetteurs, d'impulsions, ou de voies, notre algorithme n'aura pas implémenté des ressources inutiles. De plus, un autre avantage majeur de ces files d'attente est qu'il est possible, suivant la vitesse de production ou de consommation des ressources par les différents blocs, de paralléliser cette production ou cette consommation dynamiquement, afin qu'il n'y ait pas de blocage de la chaîne complète à cause d'un bloc de traitement plus lent.

Résumé des optimisations courantes (*V3*) : transformation du code C++ en OpenCL , mise en mémoire locale des variables en lecture seule utilisées fréquemment, réécriture du code en dataflow et adaptation du parallélisme suivant le débit de chaque bloc.

VII.3.3 Bilan : résultats et comparaison CPU/GPU/FPGA

Afin d'effectuer une comparaison exhaustive de nos différentes optimisations sur FPGA en OpenCL, nous avons également implémenté une version en CUDA sur GPU, qui s'inspire de la version V2 implémentée sur FPGA, où nous parallélisons toutes les boucles des différents blocs afin d'en maximiser le parallélisme.

Nous présentons donc au Tableau VII.6 les résultats :

— des trois grandes étapes d'optimisation sur FPGA présentées à la sous-section précédente (*V1, V2, V3*),

— de la version GPU tirant parti du parallélisme des boucles imbriquées,

— de la version de référence CPU.

Comme expliqué à la section VII.3.1.2, nous avons testé nos algorithmes sur trois scénarios, qui diffèrent par la complexité de la simulation correspondante (nombre d'émetteurs, signaux, ...).

VII.3.3.1 Optimisations FPGA

Si la première version fonctionnelle (*V1*) a une empreinte logique faible, son absence d'expression du parallélisme explique les faibles performances obtenues par rapport aux deux autres versions.

En ce qui concerne la *V2*, son parallélisme permet d'améliorer les performances par rapport à la *V1*, de manière significative pour les scénarios les plus complexes et donc parallélisables (Scénario 2 et 3).

Toutefois, comme expliqué à la section VII.3.2.2, nous avons parallélisé les boucles sur le nombre maximum d'itérations possibles. C'est à dire que dans un scénario avec

TABLE VII.6 – Génération d'échantillons : comparaison FPGA/GPU/CPU

Version	Utilisation empirique (%)	Fréq. (MHz)	Scénario 1 TEM[1] (s)	Scénario 1 E[1] (W*s)	Scénario 2 TEM[1] (s)	Scénario 2 E[2] (W*s)	Scénario 3 TEM[1] (s)	Scénario 3 E[2] (W*s)
CPU - E5 2667	-	2900	27.2	2366	42.3	3680	78.2	6803
GPU - 1080 Ti	-	1480	11.5	1966	14.8	2531	19.1	3247
FPGA - Xilinx XCU115								
V1 : fonctionnelle	29	287	9.4	85	21.7	195	48.4	436
V2 : déroulage des boucles imbriquées	87	227	4.5	88	5.7	111	8.7	170
V3 : flot de données	67	300	1.3	17	4.8	64	10.4	139

[1] : Temps d'exécution mesuré
[2] : Énergie

5 émetteurs sur 32 maximums, nous aurions déroulé 32 fois la boucle correspondante. C'est pourquoi, pour des scénarios avec quelques émetteurs (scénarios 1 et 2), cette optimisation est inefficace car moins rapide que la V3 tout en consommant significativement plus, à cause de la sous-utilisation du circuit électronique généré sur le FPGA.

La version *V3* dans laquelle nous avons réécrit notre code pour permettre une meilleure communication entre les différents blocs de notre programme, est la version la plus efficace sur FPGA eu égard aux performances relatives à l'utilisation logique et l'énergie consommée. Néanmoins, pour un scénario très complexe (scénario 3), la version *V2* est plus rapide que la version *V3*.

Aussi, à partir d'un certain nombre d'émetteurs, le gain en parallélisme de la version *V2* réduit les itérations inutiles des boucles, et les performances dépassent celles des autres versions. Toutefois, pour un nombre d'émetteur plus faible, l'optimisation en dataflow (*V3*) est plus performante, et elle utilise moins de ressources logiques, grâce à une adéquation algorithme architecture pertinente.

VII.3.3.2 Comparaison CPU/GPU/FPGA et conclusions

A partir d'un code CPU de référence, nous avons pu tester nos différentes optimisations sur FPGA et GPU sur trois scénarios de complexité différente, mais représentant tous une fenêtre de temps simulé de 30 secondes.

Ainsi, un temps d'exécution mesuré inférieur à 30 secondes, signifie que la suite des traitements (la synthèse numérique directe) peut être testée en temps réel à partir de cette génération d'échantillons, puisque le taux d'expansion τ est dans ce cas inférieur à 1.

Pour les scénarios 2 et 3, l'implémentation CPU de référence ne permet pas de tenir le temps réel. Par contre, les optimisations V2 et V3 sur FPGA en OpenCL et l'optimisation en CUDA sur GPU permettent d'obtenir $\tau < 1$. Cela signifie que ces implémentations peuvent en théorie alimenter en temps réel les prochains blocs de traitements.

L'avantage va tout de même à nos versions *V2* et *V3* sur FPGA. Non seulement les

temps d'exécution sur les trois scénarios sont les plus rapides de toutes les optimisations, mais ce sont également les plus efficaces énergétiquement. Cela peut s'expliquer par la granularité du parallélisme requis. En effet, il est souvent possible de paralléliser sur les différents cœurs d'un CPU d'un ordre de grandeur de 8, là où les GPUs, avec leur très grand nombre de cœurs, peuvent prendre en charge un parallélisme colossal. Mais entre les deux, la force du FPGA est de pouvoir exprimer un parallélisme paramétrable qui peut être modulé suivant le besoin et les ressources disponibles.

De plus, parce que la suite des traitements est effectuée sur FPGA, l'avantage d'avoir ces premiers blocs qui y sont également exécutés, permet une communication interne efficace, sans devoir repasser par le CPU comme pour la version GPU.

Dans ce cas d'étude, où le parallélisme réel dépend de la configuration initiale, l'approche modulaire d'un FPGA permet de tirer pleinement parti de ce parallélisme dynamique. De plus, le nombre d'itérations parallélisable maximum est de 1024 (comme expliqué en section VII.3.1.2), et les 3584 cœurs du GPU 1080 Ti seront donc largement sous-utilisés. Il serait donc intéressant de tester cet algorithme sur un GPU avec moins de cœurs, ce qui aurait pour conséquence d'en améliorer l'utilisation interne, et donc d'augmenter son efficacité énergétique.

Une fois encore, le choix de la meilleure optimisation dépend des contraintes et du contexte de l'application.

Nous avons pu voir que les deux optimisations les plus performantes sont la version *V2* (Déroulage des boucles imbriquées) et la version *V3* (réécriture du programme en dataflow) sur FPGA.

Le seul cas où il serait pertinent de choisir la version *V2* est avec un scénario complexe, et si le temps d'exécution est la contrainte prioritaire. Sinon, la version *V3* est le meilleur choix, car non seulement elle consomme moins, mais il reste suffisamment de ressources logiques pour implémenter d'autres traitements sur le même FPGA.

Si par contre la contrainte en temps de développement est plus importante que la performance brute, alors la version *V2* est la plus efficace, puisque la version *V3* nous a demandé une réécriture significative du code initial.

Ces deux optimisations ont un $\tau < 1$, et il est possible d'envisager d'améliorer la représentativité du modèle :

— soit en choisissant un pas de simulation plus fin,

— soit en augmentant le nombre d'émetteurs simultanés ou d'impulsions simultanées.

De plus, il est possible, à partir de ce modèle de référence généré avec les outils OpenCL sur le FPGA, d'alimenter directement l'étape de synthèse numérique directe sur un système réel ou simulé, et il est donc tout à fait envisageable d'utiliser ce modèle de référence également comme modèle implémenté sur cible matérielle.

Chapitre VIII

Algorithmes généraux - Benchmark (Intel)

Sommaire

VIII.1 Remarques introductives

Dans ce chapitre, nous appliquons notre méthodologie d'accélération sur sept algorithmes issus de deux suites de benchmark (MachSuite et Rodinia), présentés au Tableau VIII.1.

TABLE VIII.1 – Algorithmes optimisés en OpenCL de deux suites de Benchmark (Arria10)

MachSuite	Rodinia
AES	K-nearest
GEMM	Needleman-Wunsch
SPMV	Hybridsort
	K-means

La description précise de ces algorithmes peut être trouvée en [MachSuite, 2019] et [Rodinia, 2019], et seuls les deux premiers seront détaillés dans les deux sections suivantes.

La section VIII.4 est quant à elle consacrée à la présentation des résultats des meilleures optimisations de ces différents algorithmes en OpenCL sur l'Arria10, et à leur comparaison avec les optimisations en OpenMP sur le CPU d'Intel E5-2667 et en CUDA sur le GPU d'Nvidia 1080Ti, incluses dans les benchmarks.

VIII.2 K-nearest Neighbors (Rodinia)

VIII.2.1 Description

L'algorithme permet de trouver les *k* plus proches voisins dans une base de données aléatoires. Le principe est ici appliqué à la détection des ouragans les plus proches et illustré à la Figure VIII.1.

À partir d'une latitude $y_{latitude}$ et d'une longitude $y_{longitude}$ d'un ouragan, nous calculons la distance Euclidienne à la position courante x, donnée par la formule (VIII.1).

$$distance(x, y) = \sqrt{(x_{latitude} - y_{latitude})^2 + (x_{longitude} - y_{longitude})^2} \quad \text{(VIII.1)}$$

VIII.2.2 Caractérisation

Les caractéristiques de l'algorithme sont les suivantes :

— Le calcul de la distance entre une position fournie et tous les points du jeu de données se fait dans une seule fonction à l'aide d'une simple boucle de parcours d'un tableau.

— L'objectif de l'algorithme est de calculer, pour une série de 10'000 points, leur distance aux 100 ouragans pour en déduire les k plus proches voisins. Il faut donc faire $10'000 * 100 = 1'000'000$ calculs de distances.

— Nous n'effectuons que l'algorithme de calcul des distances sur FPGA.

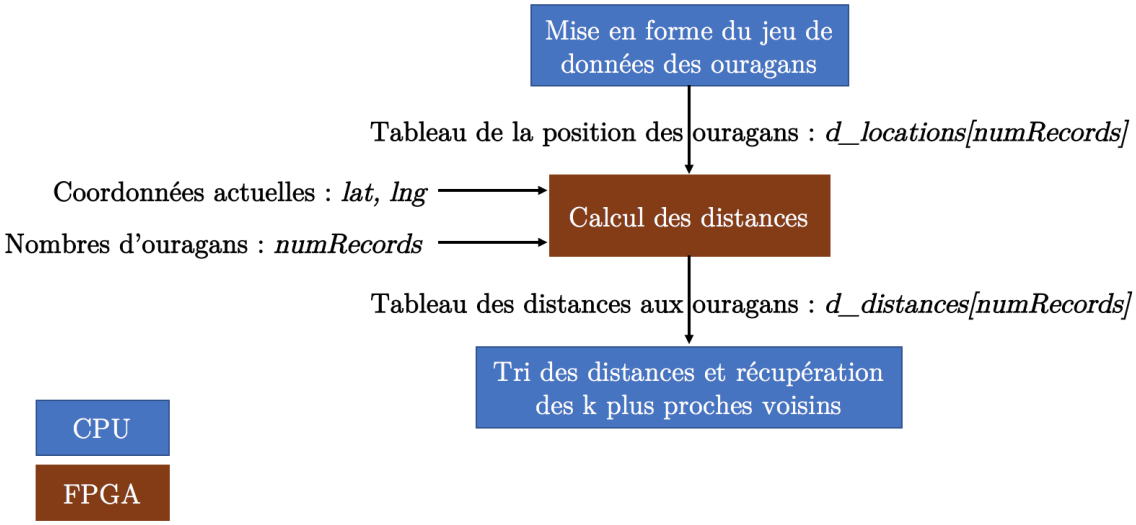


FIGURE VIII.1 – Algorithme des k plus proches voisins (CPU + FPGA).

L'intérêt d'accélérer cet algorithme est d'évaluer la performance des différents modes de parallélisations possibles sur FPGA en OpenCL, comme présentés au chapitre IV.

VIII.2.3 Application de la méthodologie

Nous retrouvons la démarche d'application de la méthodologie décrite à la section V.3, et la première implémentation fonctionnelle en OpenCL (*V1*) a les caractéristiques suivantes :

— Les variables d'entrées *lat*, *lng*, et *numRecords* sont passées en paramètres non pointés,

— Le tableau $d\_locations$ est déclaré en $\_\_global\ read\_only\ restrict$ en OpenCL.

— Le tableau $d\_distances$ est déclaré en $\_\_global\ write\_only\ restrict$ en OpenCL.

— Nous utilisons les fonctions racine et puissance $native\_sqrt$ et $native\_pow$ optimisées en OpenCL.

— Nous appliquons la vectorisation des paramètres d'entrée (Section IV.2.1.2).

— Le parallélisme exprimé est le parallélisme le plus simple, à savoir le pipeline de boucle en SWIK (Section IV.1.2.1).

Nous définissons les versions suivantes qui dérivent toutes de la version initiale *V1*, et implémentent une technique différente de parallélisation :

V2 : SWIK, déroulage de boucle (Section IV.1.2.2)

V3 : NDRK, parallélisme simple des work-items (Section II.6.2)

V4 : NDRK, réplication de pipeline élémentaire (Section IV.2.2)

V5 : NDRK, réplication de pipeline élémentaire et vectorisation des work-items (Section IV.2.1.1)

Le Tableau VIII.2 présente les résultats de ces différentes optimisations.

TABLE VIII.2 – K-means : performances des optimisations (Arria10)

Version	Utilisation empirique (%)	F (MHz)	Temps d'exécution mesuré (s)	Performance relative (ms)
V1 : SWIK, pipeline	10	197	34.87	3.49
Expression du parallélisme				
V2 : SWIK, déroulage x64	91	164	0.62	0.56
V3 : NDRK simple	8	186	37.28	2.98
V4 : NDRK, réplication PE x16	37	179	2.63	0.97
V5 : NDRK, réplication PE x16 et vectorisation x4	82	172	0.71	0.59

Nous pouvons constater que :

— l'approche SWIK est plus performante tant d'un point de vue absolu que relatif.

— l'approche NDRK est économe en ressources mais moins efficace malgré une fréquence d'utilisation plus haute.

— la vectorisation NDRK est très efficace de concert avec une réplication des pipelines élémentaires.

Ces conclusions concordent avec les bonnes pratiques énoncées au chapitre IV, qui poussent à privilégier une implémentation en SWIK pour exprimer le plus finement possible le parallélisme d'un algorithme en répliquant le moins de ressources possibles. Par contre, dans certains cas, il peut être intéressant d'envisager une approche NDRK, et nous allons l'illustrer par un cas pratique à la section suivante.

VIII.3 Needleman-Wunsch (Rodinia)

VIII.3.1 Description

Cet algorithme est une méthode d'optimisation globale pour l'alignement des séquences d'ADN. Les paires potentielles de séquences sont organisées dans une matrice 2D dite de score, comme illustré à la Figure VIII.2. En partant de la première case en haut à gauche de la matrice, l'algorithme remplit la matrice de scores [1], qui représente la valeur du chemin pondéré maximal de similitude se terminant à chaque cellule. Un processus de suivi est utilisé pour rechercher l'alignement optimal, et consiste à remonter le chemin dessiné en partant de la case en bas à droite de la matrice.

Pour calculer le score d'une case, qui consiste à des multiplications entières simples, nous n'avons besoin de connaître que les valeurs des trois voisins les plus proches respectivement à gauche, en haut, et en diagonale gauche haut, comme illustré à la Figure VIII.3.

1. Suivant différentes méthodes détaillées en [Needleman, 2013]

Needleman-Wunsch

match = 1 mismatch = -1 gap = -1

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

FIGURE VIII.2 – Matrice de score construite pour deux séquences ADN

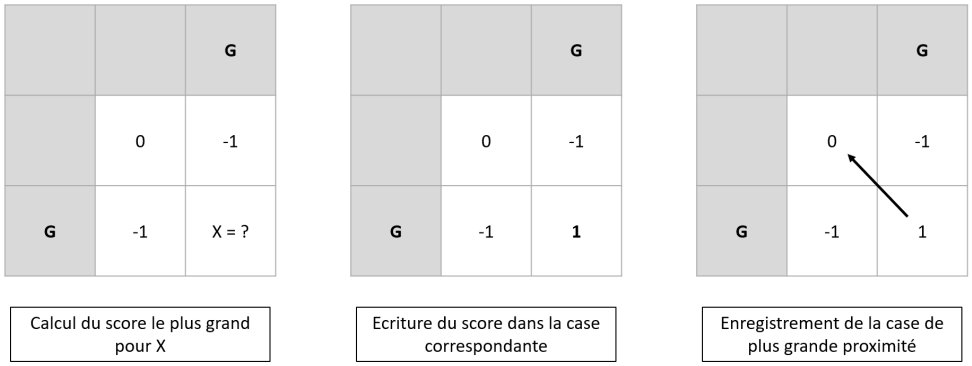


FIGURE VIII.3 – Classification d'un nouvel élément grâce au calcul des plus proches voisins.

VIII.3.2 Caractérisation

— La construction de la matrice dépend de certaines étapes précédentes.

— Il est possible d'effectuer les traitements de chaque case d'une diagonale en même temps, comme illustré à la Figure VIII.4.

— Le parcours final de recherche du chemin optimal une fois la matrice construite est séquentiel mais rapide.

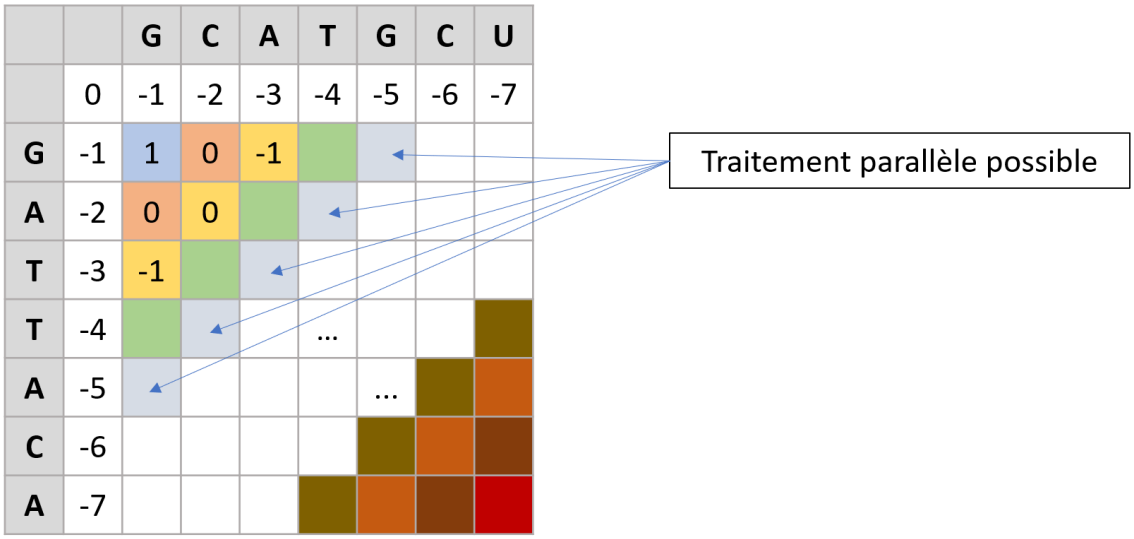


FIGURE VIII.4 – Parallélisme possible de l'algorithme de Needleman-Wunsch

VIII.3.3 Application de la méthodologie

La solution pertinente d'optimisation est donc de paralléliser sur les diagonales, et, à partir de ces différentes remarques, nous avons choisi d'implémenter les versions suivantes :

— V1 : SWIK, pipeline de boucle (version initiale)

— V2 : SWIK, déroulage de la boucle des diagonales x8

— V3 : NDRK, réplication du pipeline élémentaire x8

— V4 : NDRK, réplication du pipeline élémentaire x8, implémentation d'un cache local tampon entre deux diagonales

En ce qui concerne les versions V3 et V5, le principe de la réplication du pipeline élémentaire est le suivant : nous empilons autant de work-items qu'il y a de cases de la matrice, et nous les lançons un par un en veillant à ce que les premières diagonales soient calculées en premier. Ainsi, la charge de travail pourra se répartir de façon dynamique, et tous les cœurs de calcul seront utilisés sauf pour les premières et dernières diagonales.

Le Tableau VIII.3 résume les performances des différentes versions implémentées.

Pour cet algorithme, les versions NDRK V3 et V5 sont toutes les deux plus optimales que la version équivalente SWIK.

Dérouler une boucle en SWIK n'est efficace que si le nombre d'itérations de la boucle déroulée est fixe. Or, dans notre cas, chaque diagonale a une taille différente. L'approche NDRK est par contre particulièrement adaptée, puisqu'il est possible dynamiquement d'équilibrer la charge des différents cœurs de calcul que constituent les pipelines élémentaires. De plus, une implémentation efficace d'une structure mémoire permet d'augmenter les performances de notre optimisation en permettant à tous les work-items d'accéder à une zone mémoire locale rapide grâce à une fenêtre glissante sur les diagonales.

TABLE VIII.3 – Needleman-Wunsch : performances des optimisations (Arria10)

Version	Utilisation empirique (%)	F (MHz)	Temps d'exécution mesuré (s)	Performance relative (ms)
V1 : SWIK, pipeline	29	197	117	33.93
V2 : SWIK, déroulage x8	77	167	25.14	19.36
V3 : NDRK, réplication PE x8	75	192	26.31	19.73
V4 : NDRK, réplication PE x8 et cache local	86	185	9.70	8.34

VIII.4 Bilan

VIII.4.1 Notion d'optimisation efficace sur FPGA

Des différents algorithmes implémentés sur FPGA, l'optimisation la plus pertinente pour chacun d'eux était :

— K-nearest (KN) : SWIK, déroulage de boucle et vectorisation des paramètres d'entrée.

— Needleman-Wunsch (NW) : NDRK, réplication du Pipeline élémentaire (Parallélisation dynamique en fonction de la charge de travail) et fenêtre glissante.

— Hybridsort (HS) : SWIK, pipeline, réécriture de l'algorithme (Implémentation complète de l'arbre de comparaison sur FPGA).

— AES (AES) : Déroulage de boucle et communication inter-kernels (files FIFO).

— GEMM (GEMM) : NDRK et utilisation de la mémoire locale partagée entres work-items.

— SPMV (SPMV) : SWIK, déroulage de boucle et utilisation d'un cache en mémoire locale.

— K-means (KM) : SWIK, déroulage de boucle et vectorisation des paramètres d'entrée.

VIII.4.2 Résultats et comparaison avec les GPUs

Le Tableau VIII.5 compare le temps d'exécution mesuré et l'énergie consommée par les différentes implémentations optimisées en OpenCL pour le FPGA, en OpenMP pour le CPU, et en CUDA pour deux GPUs, un bureautique (1080 Ti), l'autre embarqué (Jetson TX2).

La comparaison entre le FPGA et le CPU montre que toutes les implémentations OpenCL optimisées, sauf une, sont plus efficaces que les équivalents CPU. Par contre, seulement deux algorithmes sont plus efficaces sur FPGA que sur GPU bureautique, en l'occurence nous retrouvons l'algorithme de Needleman-Wunsch présenté à la section VIII.3.

Algorithmes \ Archi	Ratios des implémentations optimisées $\frac{T_{Archi}}{T_{ARRIA10}}$			$\frac{E_{ARRIA10}}{E_{Archi}}$ (%)			Dimension de l'algorithme
	CPU OpenMP	GPUe CUDA	GPUb CUDA	CPU OpenMP	GPUe CUDA	GPUb CUDA	
KN	3,70	2,04	0,17	9,15	37,27	28,93	1024*10
NW	47,93	37,2	22,40	1,05	5,74	0,43	512*512
HS	4,79	4,5	1,74	5,80	19,53	2,40	5500
AES	22,71	9,5	0,89	1,60	42,18	5,40	4000
GEMM	11,46	0,67	0,24	3,51	79,27	32,28	512*512
KM	0,87	1,8	0,65	41,97	29,21	10,70	1024*10
SPMV	3,24	0,97	0,14	9,92	93,5	36,06	512*512

FPGA : Intel Arria10/Xilinx Kintex KCU115 CPU : Intel E5-2667 GPUe : Nvidia Jetson TX2 GPUb : Nvidia 1080Ti

FIGURE VIII.5 – Performances des meilleurs optimisations OpenCL et comparaison CPU/GPU bureautique/GPU embarqué/FPGA

Le GPU gère mal le parallélisme dynamique, et l'efficacité d'une optimisation sur cette architecture passe par l'utilisation simultanée de tous les cœurs, ce qui est impossible au vu du motif de calcul de l'algorithme.

Si on compare maintenant l'efficacité de notre meilleure implémentation OpenCL à celle obtenue en CUDA sur GPU embarqué, le FPGA est plus rapide dans six cas sur hut que le GPU embarqué. De plus, le FPGA a, dans tous les cas, une meilleure efficacité énergétique que les trois autres architectures.

En conclusion, nous avons pu voir que l'utilisation des outils OpenCL permettait dans certains cas d'obtenir un facteur d'accélération significatif par rapport au CPU mais aussi au GPU dans certains cas. Nous retrouvons donc avec cet outil l'efficacité énergétique des FPGAs.

Par contre, notre démarche d'adéquation algorithme architecture est essentielle, puisque les versions initiales des algorithmes OpenCL étaient presque toutes moins performantes que la version CPU.

Conclusion générale : limites et perspectives de la recherche

Notre contribution a été d'évaluer les outils de description haut niveau en OpenCL des FPGAs, et de fournir une méthodologie exhaustive basée sur une approche logicielle des optimisations efficaces pour l'adéquation algorithme architecture sur FPGA. En ce sens nous avons développé une approche permettant de réduire la rupture entre les différents métiers logiciels et matériels.

En partant du constat que l'industrie des semi-conducteurs avait des difficultés croissantes à accroître les performances des architectures homogènes comme les CPUs, nous avons, **dans la première partie**, présenté différents relais de croissance pour l'accélération d'algorithmes, comme l'étude des ordinateurs quantiques, ou encore la superposition 3D de puces classiques. Mais, la piste d'intérêt que nous avons développée dans ces travaux est la démarche d'adéquation algorithme architecture.

En particulier, les récents changements technologiques dans les FPGAs, comme l'augmentation du nombre de leurs DSPs ou l'intégration de nouvelles technologies mémoires, les placent de nouveau en compétition face aux CPUs et aux GPUs, et c'est pourquoi nous avions choisi de nous intéresser à cette technologie comme une alternative viable aux CPUs et GPUs traditionnels pour l'accélération d'algorithmes.

Dans la deuxième partie, nous avons proposé un modèle de premier ordre d'un FPGA comme co-processeur couplé à un CPU. Nous avons pu l'enrichir de différentes métriques qui permettent de caractériser l'efficacité et la performance d'une implémentation FPGA. Puis, nous avons pu détailler et caractériser l'efficacité d'un certain nombre d'optimisations OpenCL, pour finalement les mettre en forme dans une méthodologie d'accélération d'algorithmes dans une démarche d'optimisation de Pareto multicritères. Son intérêt majeur est de fournir, en s'appuyant sur les différentes métriques établies, une projection rapide des performances d'une implémentation sans avoir à générer le Bistream, ce qui a pour conséquence de réduire significativement le temps de compilation.

Dans la troisième partie, nous avons appliqué notre méthodologie sur un grand nombre d'algorithmes aux caractéristiques variées, et avons pu obtenir des gains d'accélération substantiels suivant le type d'algorithme considéré. La Figure VIII.6 illustre, sur nos dix algorithmes présentés, les performances relatives de chacune des architectures, tant en terme de temps d'exécution que d'efficacité énergétique.

Dans le domaine de la tomographie, le pipeline élémentaire généré par l'outil OpenCL d'Intel est plus efficace que les pipelines correspondants sur les CPUs et GPUs, mais reste moins efficace qu'une implémentation équivalente en VHDL. L'inadéquation entre

	Temps d'exécution (Le plus rapide)			Consommation énergétique (Le plus efficient)		
	1	2	3	1	2	3
Tomographie	GPU	FPGA	CPU	GPU	FPGA	CPU
Radar	GPU	FPGA	CPU	FPGA	GPU	CPU
Générateur	FPGA	GPU	CPU	FPGA	GPU	CPU
KN	GPU	FPGA	CPU	FPGA	GPU	CPU
NW	FPGA	GPU	CPU	FPGA	CPU	GPU
HS	FPGA	GPU	CPU	FPGA	CPU	GPU
AES	GPU	FPGA	CPU	FPGA	GPU	CPU
GEMM	GPU	FPGA	CPU	FPGA	GPU	CPU
SPMV	GPU	CPU	FPGA	FPGA	CPU	GPU
KM	GPU	FPGA	CPU	FPGA	GPU	CPU

FIGURE VIII.6 – Indicateurs de performances des trois architectures d'intérêt sur les dix algorithmes présentés

le caractère massivement parallèle de l'algorithme, les accès irréguliers aux données en mémoire globale, et la technologie mémoire supérieure des GPUs actuels explique, en plus d'une fréquence d'utilisation faible, pourquoi le temps d'exécution mesuré sur GPU ainsi que l'efficacité énergétique de l'optimisation CUDA sont meilleures que ceux du FPGA.

Par contre, l'implémentation optimisée de notre génération de signaux numériques superhétérodynes est non seulement la plus rapide par rapport aux optimisations CPUs et GPUs, mais, quelque soit le scénario, elle tient également le temps réel, ce qui peut permettre d'augmenter la représentativité du générateur de signaux.

Sur la grande majorité de nos algorithmes, les outils parviennent à conserver l'une des propriétés principales des FPGAs, à savoir leur efficacité énergétique. Toutefois, la promesse de pouvoir programmer ces architectures que du point de vue du logiciel n'est que partiellement tenue. En effet, même si nous n'avons utilisé que le standard OpenCL pour nos implémentations FPGAs, nous avons dû réécrire plus de la moitié de nos algorithmes pour les adapter aux spécificités des architectures FPGAs.

Si Intel a fait le choix de pousser assez loin la compatibilité de ses outils avec le standard OpenCL, Xilinx s'en éloigne progressivement. Et pourtant, l'entreprise a dévoilé récemment sa nouvelle architecture, Versal, et a longuement insisté sur la possibilité d'en tirer parti à de très nombreux étages d'abstraction, comme l'illustre la Figure VIII.7.

Plateforme de calcul hétérogène, embarquant de la logique programmable, des processeurs temps réels, des processeurs classiques multi-coeurs, ainsi que du matériel spécialisé pour le domaine de l'intelligence artificielle, cette puce aura une finesse de gravure de 7nm, pour une disponibilité au premier trimestre 2020.

Dans un domaine en plein essor, où une nouvelle architecture est présentée tous les trimestres, à l'image des puces spécialisées de Tesla pour la conduite autonome, des TPUs de Google, ou encore des TensorCore chez Nvidia, le pari de Xilinx est risqué, mais, s'il s'avère payant, a le potentiel de significativement impacter l'industrie des semi-conducteurs.

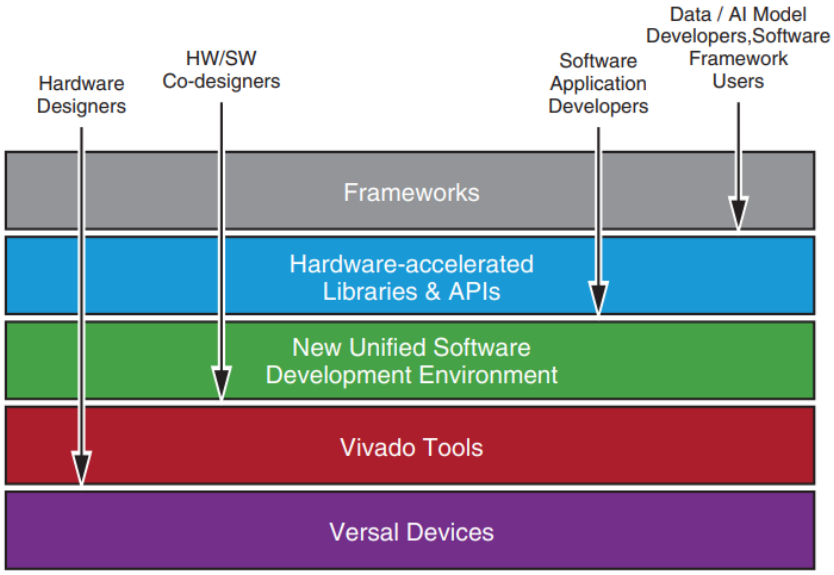


FIGURE VIII.7 – Architecture VERSAL : une variété d'outils de programmation

Source: Xilinx

Les outils haut niveau actuels ne permettent toujours pas à l'heure actuelle de rivaliser avec leurs équivalents bas niveau, mais leurs performances restent néanmoins en nette évolution et il est possible qu'à terme, les langages de description matériel disparaissent comme l'assembleur autrefois, pour laisser place à des langages haut niveau efficaces. Dans notre algorithme de génération d'échantillon, nous avons pu obtenir une implémentation qui était non seulement plus efficace énergétiquement et plus rapide que les autres architectures, mais pouvait également s'interfacer avec d'autres traitements FPGAs pour générer des signaux en temps réel à partir d'une description OpenCL.

Dans la dynamique des travaux effectués dans le domaine de la reconstruction tomographique, il nous semble pertinent d'étudier la classe des projecteurs ray-driven [Nguyen and Lee, 2015]. En effet, ce type d'algorithme a été délaissé ces dernières années parce que les multiples branchements conditionnels occasionnaient trop de divergences sur les architectures GPUs. Potentiellement efficace sur les architectures FPGAs, il s'agirait alors d'évaluer cet algorithme en appliquant notre méthodologie pour l'optimiser en OpenCL.

Les années actuelles et à venir sont un nouvel âge d'or pour l'industrie des semiconducteurs, et l'hétérogénéité des architectures constitue un levier de croissance essentiel. Aussi, un autre axe pertinent de recherche serait d'explorer, dans le contexte de l'accélération d'algorithmes, les architectures Versal de Xilinx, ou encore les CPUs à base de RISC-V qui promettent d'apporter d'insuffler un nouveau souffle aux CPUs qui avaient ces derniers temps du mal à se réinventer.

Publications

REVUES INTERNATIONALES

- **3D Tomography back-projection parallelization on Intel FPGAs using OpenCL**
  M. Martelli, N. Gac, A. Mérigot, C. Enderli
  Springer Journal of Signal Processing Systems *n°91 vol 7 p 731-743 2019*

CONFÉRENCES INTERNATIONALES

- **GPU Acceleration : OpenACC for Radar Processing Simulation**
  M. Martelli, C. Enderli, N. Gac, A. Vermesse, A. Mérigot
  IEEE International RADAR Conference (RADAR)
  Toulon, France, Septembre 2019
- **3D Tomography back-projection parallelization on FPGAs using OpenCL**
  M. Martelli, N. Gac, A. Mérigot, C. Enderli
  Design and Architecture for Signal and Image Processing (DASIP)
  Dresden, Germany, Septembre 2017
- **An OpenCL design for tomographic reconstruction on FPGA**
  M. Martelli, M. Seznec, N. Heemeryck
  International Conference on Field-Programmable Logic and Applications (FPL)
  Gant, Belgique, Septembre 2017

CONFÉRENCES NATIONALES

- **Accélération sur GPU d'une simulation radar avec OpenACC**
  M. Martelli, C. Enderli, N. Gac, A. Vermesse, A. Mérigot
  XXVIIème Colloque francophone de traitement du signal et des images (GRETSI)
  Lille, France, Août 2019
- **Harnessing FPGAs potential with OpenCL**
  M. Martelli
  Colloque du GDR System On Chip - System In Package (SoC SIP)
  Paris, France, Juin 2018

DISTINCTIONS

- **Lauréat du concours Innovate Europe Design Contest 2017**
  Catégorie HPC
  Co-organisé par INTEL - Terasic - CNFM

Bibliographie

[Iec, 2015] (2015). The International Technology Roadmap For Semiconductors 2.0. *Semiconductor Industry Association*.

[Int, 2017] (2017). Intel FPGA SDK for OpenCL Programming Guide. *Intel*.

[Alibaba, 2019] Alibaba (2019). XT910 fastest RISC-V CPU : 16 cores @ 2.5 GHz. *Alibaba Cloud Summit*.

[Asanović and Patterson, 2014] Asanović, K. and Patterson, D. A. (2014). Instruction Sets Should Be Free : The Case For RISC-V.

[Aspray, 1997] Aspray, W. (1997). The intel 4004 microprocessor : what constituted invention ? *IEEE Annals of the History of Computing*, 19(3) :4–15.

[Babbage, Charles, 1851] Babbage, Charles (1851). *The Exposition of 1851*. John Murray, London.

[Bardeen et al., 1948] Bardeen, J., Shockley, W., and Brattain, W. (1948). Technical Memoranda Transistor. https ://www.smecc.org/bell_labs_holding_page.htm.

[Black, 2019] Black, D. (2019). Xilinx vs. Intel : FPGA Market Leaders Launch Server Accelerator Cards. *HPC wire*.

[Bohr, 2007] Bohr, M. (2007). A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1) :11–13.

[Bollaert, 2008] Bollaert, T. (2008). *Catapult Synthesis : A Practical Introduction to Interactive C Synthesis*, pages 29–52. Springer Netherlands, Dordrecht.

[Cadenelli et al., 2019] Cadenelli, N., Jaksić, Z., Polo, J., and Carrera, D. (2019). Considerations in using opencl on gpus and fpgas for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94 :148 – 159.

[Canis et al., 2011] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). Legup : High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA. ACM.

[Cellania, 2015] Cellania, M. (2015). Ada Lovelace, the Enchantress of Numbers. http ://mentalfloss.com/article/53131/ada-lovelace-first-computer-programmer.

[Chapdelaine, 2019] Chapdelaine, C. (2019). *Bayesian iterative reconstruction methods for 3D X-ray Computed Tomography*. PhD thesis. Thèse de doctorat dirigée par Soussen, Charles Traitement du signal et des images Paris Saclay 2019.

[Christian Wagner and Noreen Harned, 2010] Christian Wagner and Noreen Harned (2010). Lithography gets extreme. *Nature Photonics*.

[Coussy et al., 2008] Coussy, P., Chavet, C., Bomel, P., Heller, D., Senn, E., and Martin, E. (2008). *GAUT : A High-Level Synthesis Tool for DSP Applications*, pages 147–169. Springer Netherlands, Dordrecht.

[Craig Gidney and Martin Ekerå, 2019] Craig Gidney and Martin Ekerå (2019). How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *arXiv.org*.

[Czajkowski et al., 2012] Czajkowski, T. S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., and Singh, D. P. (2012). From opencl to high-performance hardware on fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534.

[Danowitz et al., 2012] Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., and Horowitz, M. (2012). Cpu db : Recording microprocessor history. *Commun. ACM*, 55(4) :55–63.

[D'Arcy et al., 2019] D'Arcy, G., Márquez-Grant, N., and Lane, D. W. (2019). Baggage scanners and their use as an imaging resource in mass fatality incidents. *International Journal of Legal Medicine*.

[DeBenedictis et al., 2018] DeBenedictis, E. P., Humble, T. S., and Gargini, P. A. (2018). Quantum computer scale-up. *Computer*, 51(10) :86–89.

[DiCecco et al., 2016] DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G., and Areibi, S. (2016). Caffeinated fpgas : Fpga framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268.

[Dilger, 2017] Dilger, D. E. (2017). The new apple a11 chip with neural engine. *AppleInsider*.

[Dixit, 1993] Dixit, K. M. (1993). Overview of the spec benchmarks. In *The Benchmark Handbook*.

[Emmerich and Deutz, 2018] Emmerich, M. T. M. and Deutz, A. H. (2018). A tutorial on multiobjective optimization : fundamentals and evolutionary methods. *Natural Computing*, 17(3) :585–609.

[Feldkamp et al., 1984] Feldkamp, L. A., Davis, L. C., and Kress, J. W. (1984). Practical cone-beam algorithm. *J. Opt. Soc. Am. A*, 1(6) :612–619.

[Frank et al., 2001] Frank, D. J., Dennard, R. H., Nowak, E., Solomon, P. M., Taur, Y., and Hon-Sum Philip Wong (2001). Device scaling limits of si mosfets and their application dependencies. *Proceedings of the IEEE*, 89(3) :259–288.

[Gac and Djafari, 2014] Gac, N. and Djafari, A.-M. (2014). opgpuTomoGPI - Ref CNRS du pré-dépôt APP 11562-01 (num IDDN prochainement disponible).

[Gac et al., 2008] Gac, N., Mancini, S., Desvignes, M., and Houzet, D. (2008). High Speed 3D Tomography on CPU, GPU, and FPGA. *EURASIP journal on Embedded Systems*.

[Garcia et al., 2006] Garcia, P., Compton, K., Schulte, M., Blem, E., and Fu, W. (2006). An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*.

[Geyer et al., 2015] Geyer, L. L., Schoepf, U. J., Meinel, F. G., Nance, J. W., Bastarrika, G., Leipsic, J. A., Paul, N. S., Rengo, M., Laghi, P. A., and Cecco, C. N. D. (2015). State of the Art : Iterative CT Reconstruction Techniques. *Journal of Food Processing & Technology*.

[Gordon Earle Moore, 1965] Gordon Earle Moore (1965). Cramming More Components onto Integrated Circuits. *Electronics*.

[Heigl and Kowarschik, 2007] Heigl, B. and Kowarschik, M. (2007). High-speed reconstruction for c-arm computed tomography. In *Proc. of Fully 3D*, pages 25–28.

[Hendriks et al., 2011] Hendriks, M., Geilen, M., and Basten, T. (2011). Pareto analysis with uncertainty. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 189–196.

[Herbert Mataré and Heinrich Welker, 1948] Herbert Mataré and Heinrich Welker (1948). Crystal Device for Controlling Electric Currents by Means of a Solid Semiconductor. *Brevet FRD1010427*.

[Iain Goddard, 2002] Iain Goddard, M. T. (2002). High-speed cone-beam reconstruction : an embedded systems approach.

[IBM, 2019] IBM (2019). IBM Q System One. https ://newsroom.ibm.com/2019-01-08-IBM-Unveils-Worlds-First-Integrated-Quantum-Computing-System-for-Commercial-Use.

[Jakob and Blume, 2014] Jakob, W. and Blume, C. (2014). Pareto optimization or cascaded weighted sum : A comparison of concepts. *Algorithms*.

[Jimenez and Lin, 2001] Jimenez, D. A. and Lin, C. (2001). Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206.

[KhronosGroup, 2019] KhronosGroup (2019). Spécification du standard OpenCL V2.2. https ://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html.

[Kim et al., 2012] Kim, J. K., Fessler, J. A., and Zhang, Z. (2012). Forward-projection architecture for fast iterative image reconstruction in x-ray ct. *IEEE Transactions on Signal Processing*, 60(10) :5508–5518.

[Leeser et al., 2005] Leeser, M., Coric, S., Miller, E., Yu, H., and Trepanier, M. (2005). Parallel-beam backprojection : An fpga implementation optimized for medical imaging. *Journal of VLSI signal processing systems for signal, image and video technology*, 39(3) :295–311.

[Limaye and Adegbija, 2018] Limaye, A. and Adegbija, T. (2018). A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158.

[Lu et al., 2001] Lu, H., Cheng, J.-H., Han, G., Li, L., and Liang, Z. (2001). A 3D distance-weighted Wiener filter for Poisson noise reduction in sinogram space for SPECT imaging. *Medical Imaging, Physics of Medical Imaging*.

[MachSuite, 2019] MachSuite (2019). Benchmarks for Accelerator Design and Customized Architectures. https ://breagen.github.io/MachSuite/.

[Mancini and Eveno, 2004] Mancini, S. and Eveno, N. (2004). An IIR based 2D adaptive and predictive cache for image processing. In *XIX conference on Design of Circuits and Integrated Systems (DCIS'2004)*, Bordeaux, France.

[Mark D. Hill and Michael R. Marty, 2008] Mark D. Hill and Michael R. Marty (2008). Amdahl's Law in the Multicore Era. *IEEE Computer Society*.

[Martelli et al., 2019a] Martelli, M., Enderli, C., Gac, N., Vermesse, A., and Mérigot, A. (2019a). GPU Acceleration : OpenACC for Radar Processing Simulation. *IEEE International Radar Conference (RADAR2019)*.

[Martelli et al., 2019b] Martelli, M., Gac, N., Mérigot, A., and Enderli, C. (2019b). 3D Tomography Back-Projection Parallelization on Intel FPGAs Using OpenCL. *Springer Journal of Signal Processing Systems*, 91(7) :731–743.

[Microsemi, 2018] Microsemi (2018). Industry's first RISC-V SoC FPGA Architecture for Low Power Real-Time Linux.

[Mutlu, 2019] Mutlu, O. (2019). Rowhammer and beyond. In Polian, I. and Stöttinger, M., editors, *Constructive Side-Channel Analysis and Secure Design*, pages 3–12, Cham. Springer International Publishing.

[Needleman, 2013] Needleman (2013). Needleman–Wunsch algorithm : constructing the grid. https ://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm.

[Nguyen and Lee, 2015] Nguyen, V. and Lee, S. (2015). Parallelizing a matched pair of ray-tracing projector and backprojector for iterative cone-beam ct reconstruction. *IEEE Transactions on Nuclear Science*, 62(1) :171–181.

[Niklaus Wirth, 1995] Niklaus Wirth (1995). A Plea for Lean Software. *Cybersquare*.

[Nvidia, 2006] Nvidia (2006). CUDA Accelerated Computing. https ://developer.nvidia.com/cuda-zone.

[OpenACC, 0] OpenACC (0). OpenACC Specification. https ://www.openacc.org/specification.

[OpenMP, 0] OpenMP (0). OpenMP Specification. https ://www.openmp.org/.

[Pareto, Vilfredo, 1909] Pareto, Vilfredo (1909). *Manuel d'économie politique*. V. Giard and E. Brière, London.

[Patterson and Hennessy, 2007] Patterson, D. A. and Hennessy, J. L. (2007). *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Pelcat et al., 2014] Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J., and Aridhi, S. (2014). Preesm : A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 36–40.

[Putnam et al., 2008] Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., and Eggers, S. (2008). Chimps : A c-level compilation flow for hybrid cpu-fpga architectures. In *2008 International Conference on Field Programmable Logic and Applications*, pages 173–178.

[Ravi et al., 2019] Ravi, M., Sewa, A., T. G., S., and Sanagapati, S. S. S. (2019). Fpga as a hardware accelerator for computation intensive maximum likelihood expectation maximization medical image reconstruction algorithm. *IEEE Access*, 7 :111727–111735.

[ROCCC, 2013] ROCCC (2013). Overview of ROCCC 2.0 : a C to VHDL compilation toolset. http ://roccc.cs.ucr.edu.

[Rodinia, 2019] Rodinia (2019). Rodinia : Accelerating Compute-Intensive Applications with Accelerators. https ://rodinia.cs.virginia.edu/.

[Scherl et al., 2007] Scherl, H., Keck, B., Kowarschik, M., and Hornegger, J. (2007). Fast gpu-based ct reconstruction using the common unified device architecture (cuda). In *2007 IEEE Nuclear Science Symposium Conference Record*, volume 6, pages 4464–4466.

[Seznec, 2006] Seznec, A. (2006). A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism*.

[Shagrithaya et al., 2013] Shagrithaya, K., Kępa, K., and Athanas, P. (2013). Enabling development of opencl applications on fpga platforms. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 26–30.

[Shi et al., 2019] Shi, R., Wong, J., and So, H. (2019). High-Throughput Line Buffer Microarchitecture for Arbitrary Sized Streaming Image Processing. *Journal of Imaging*, 5(3) :34.

[Silva et al., 2013] Silva, B. D., Braeken, A., D'Hollander, E. H., and Touhafi, A. (2013). Performance modeling for fpgas : Extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013.

[Sivakumar M et al., 2015] Sivakumar M, S., M, B., and S, A. (2015). Ijeceierd - design of low power high performance 16-point 2-parallel pipelined fft architecture.

[Smith, 1981] Smith, J. E. (1981). A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA. IEEE Computer Society Press.

[TIMA, 2012] TIMA (2012). AUGH : Autonomous and User Guided High-level synthesis. http ://tima.imag.fr/sls/research-projects/augh/.

[Tomasulo, 1967] Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1) :25–33.

[Waidyasooriya et al., 2017] Waidyasooriya, H. M., Takei, Y., Tatsumi, S., and Hariyama, M. (2017). Opencl-based fpga-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems*, 28(5) :1390–1402.

[Wang et al., 2008] Wang, G., Yu, H., and De Man, B. (2008). An outlook on x-ray ct research and development. *Medical Physics*, 35(3) :1051–1064.

[Wegrzyn, 2001] Wegrzyn, M. (2001). FPGA-Based Logic Controllers for Safety Critical Systems. *IFAC Conference on New Technologies for Computer Control*.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline : An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[Xilinx, 2018] Xilinx (2018). SDAccel Programmers guide 2018.2. https ://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1277-sdaccel-programmers-guide.pdf.

[Xu et al., 2010] Xu, J., Subramanian, N., Alessio, A., and Hauck, S. (2010). Impulse c vs. vhdl for accelerating tomographic reconstruction. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 171–174.

[Yali, 2014] Yali, M. P. (2014). FPGA-Roofline : An Insightful Model for FPGA-based Hardware Accelerators in Modern Embedded Systems. Master's thesis, Virginia Polytechnic Institute and State University.

[Zeng, 2010] Zeng, G. (2010). *Medical image reconstruction : A conceptual tutorial*.

ÉCOLE DOCTORALE

Sciences et technologies de l'information et de la communication (STIC)

Titre : Approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA. Application à la qualification des radars et des systèmes d'écoute électromagnétique

Mots clés : Adéquation Algorithme Architecture, Radar, OpenCL, FPGA, GPU, Calcul Haute Performance

Résumé :
A l'heure où l'industrie des semi-conducteurs fait face à des difficultés majeures pour entretenir une croissance en berne, les nouveaux outils de synthèse de haut niveau repositionnent les FPGAs comme une technologie de premier plan pour l'accélération matérielle d'algorithmes face aux clusters à base de CPUs et GPUs.
Mais en l'état, pour un ingénieur logiciel, ces outils ne garantissent pas, sans expertise du matériel sous-jacent, l'utilisation de ces technologies à leur plein potentiel. Cette particularité peut alors constituer un frein à leur démocratisation.
C'est pourquoi nous proposons une méthodologie d'accélération d'algorithmes sur FPGA. Après avoir présenté un modèle d'architecture haut niveau de cette cible, nous détaillons différentes optimisations possibles en OpenCL, pour finalement définir une stratégie d'exploration pertinente pour l'accélération d'algorithmes sur FPGA.
Appliquée sur différents cas d'étude, de la reconstruction tomographique à la modélisation d'un brouillage aéroporté radar, nous évaluons notre méthodologie suivant trois principaux critères de performance : le temps de développement, le temps d'exécution, et l'efficacité énergétique.

Title : High-Level Approach for the Acceleration of Algorithms on CPU/GPU/FPGA Heterogeneous Architectures. Application to Radar Qualification and Electromagnetic Listening Systems

Keywords : Algorithm architecture co-design, Radar, OpenCL, FPGA, GPU, High Performance Computing

Abstract :
As the semiconductor industry faces major challenges in sustaining its growth, new High-Level Synthesis tools are repositioning FPGAs as a leading technology for algorithm acceleration in the face of CPU and GPU-based clusters.
But as it stands, for a software engineer, these tools do not guarantee, without expertise of the underlying hardware, that these technologies will be harnessed to their full potential. This can be a game breaker for their democratization.
From this observation, we propose a methodology for algorithm acceleration on FPGAs. After presenting a high-level model of this architecture, we detail possible optimizations in OpenCL, and finally define a relevant exploration strategy for accelerating algorithms on FPGA.
Applied to different case studies, from tomographic reconstruction to the modelling of an airborne radar jammer, we evaluate our methodology according to three main performance criteria : development time, execution time, and energy efficiency.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France