UNIVERSITE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour l'obtention du grade de

Docteur en Sciences
de l'Université Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenue par

Mohamed Amine BERGACH

Adaptation du calcul de la Transformée de Fourier Rapide sur une architecture mixte CPU/GPU intégrée

Thèse dirigée par Robert DE SIMONE

soutenue le 2 Octobre 2015

Jury :

M. Olivier SENTIEYS	Examinateur, Directeur de recherche Inria (Université de Rennes)
M. Jean-François MÉHAUT	Rapporteur, Professeur (Université de Grenoble 1)
M. Albert COHEN	Rapporteur, Directeur de recherche Inria (ENS Paris)
M. Robert DE SIMONE	Directeur de thèse, Directeur de recherche Inria Sophia Antipolis
M. Serge TISSOT	Encadrant de thèse entreprise, Technical Fellow Kontron
M. Michel SYSKA	co-Encadrant, Maître de conférences Université Nice-Sophia Antipolis

À mes parents :
Mon père qui m'a appris la discipline
Ma mère qui m'a appris la patience

"The Free Lunch Is Over"

Herb Sutter

Remerciements

Je tiens à remercier en premier lieu Robert de Simone d'avoir accepté d'être le directeur de cette thèse.

Mes remerciements vont conjointement et tout particulièrement à Serge Tissot et Michel Syska pour leur encadrement, leur soutien et leur confiance qui m'ont permis de mener à bien ce projet.

Je remercie également les rapporteurs de cette thèse Albert Cohen et Jean-François Méhaut de s'être portés volontaires à la lecture de mon mémoire. Merci également aux autres membres du jury.

Je tiens à remercier une dernière fois Robert de Simone, Michel Syska et Serge Tissot pour leur relecture de mes travaux et de m'avoir soumis leurs corrections tant sur la forme que sur mon français parfois approximatif.

Enfin, je tiens à montrer ma gratitude à tout le soutien apporté par mes proches, ma famille, plus particulièrement mes parents pour leurs encouragements ; sans oublier mes amis qui ont su me motiver en s'informant régulièrement de l'avancement de mon travail.

Cette thèse n'aurait sûrement pas abouti sans cet entourage qui a su apporter, chacun à leur manière, une aide réconfortante et précieuse.

Merci.

Résumé

Les architectures multi-cœurs *Intel Core* (IvyBridge, Haswell,...) contiennent à la fois des cœurs CPU généralistes (4), mais aussi des cœurs dédiés GPU embarqués sur cette même puce (16 et 40 respectivement). Dans le cadre de l'activité de la société Kontron (qui participe à ce financement de nature CIFRE) un objectif important est de calculer efficacement sur cette architecture des tableaux et séquences de transformées de Fourier rapides (FFT), comme par exemple on en trouve dans des applications radar. Alors que des bibliothèques natives (mais propriétaires) existent chez Intel pour les CPU, rien de tel n'est actuellement disponible pour la partie GPU.

L'objectif de la thèse était donc de définir le placement efficace de modules FFT, en étudiant au niveau théorique la forme optimale permettant de regrouper des étages de calcul d'une telle FFT en fonction de la localité des données sur un cœur de calcul unique. Ce choix a priori permet d'espérer une efficacité des traitements, en ajustant la taille de la mémoire disponible à celles des données nécessaires.

Ensuite la multiplicité des cœurs reste exploitable pour disposer plusieurs FFT calculées en parallèle, sans interférence (sauf contention du bus entre CPU et GPU). Nous avons obtenu des résultats significatifs, tant au niveau de l'implantation d'une FFT (1024 points) sur un cœur CPU SIMD, exprimée en langage C, que pour l'implantation d'une FFT de même taille sur un cœur GPU SIMT, exprimée alors en OpenCL. De plus nos résultats permettent de définir des règles pour synthétiser automatiquement de telles solutions, en fonction uniquement de la taille de la FFT son nombre d'étages plus précisément), et de la taille de la mémoire locale pour un cœur de calcul donné. Les performances obtenues sont supérieures à celles de la bibliothèque native Intel pour CPU), et démontrent un gain important de consommation sur GPU. Tous ces points sont détaillés dans le document de thèse. Ces résultats devraient donner lieu à exploitation au sein de la société Kontron.

Abstract

Multicore architectures Intel Core (IvyBridge, Haswell...) contain both general purpose CPU cores (4) and dedicated GPU cores embedded on the same chip (16 and 40 respectively). As part of the activity of Kontron (the company partially funding this CIFRE scholarship), an important objective is to efficiently compute arrays and sequences of fast Fourier transforms (FFT) such as one finds in radar applications, on this architecture. While native (but proprietary) libraries exist for Intel CPU, nothing is currently available for the GPU part.

The aim of the thesis was to define the efficient placement of FFT modules, and to study theoretically the optimal form for grouping computing stages of such FFT according to data locality on a single computing core. This choice should allow processing efficiency, by adjusting the memory size available to the required application data size. Then the multiplicity of cores is exploitable to compute several FFT in parallel, without interference (except for possible bus contention between the CPU and the GPU). We have achieved significant results, both in the implementation of an FFT (1024 points) on a SIMD CPU core, expressed in C, and in the implementation of a FFT of the same size on a GPU SIMT core, then expressed in OpenCL.

In addition, our results allow to define rules to automatically synthesize such solutions, based solely on the size of the FFT (more specifically its number of stages), and the size of the local memory for a given computing core. The performances obtained are better than the native Intel library for CPU, and demonstrate a significant gain in consumption on GPU. All these points are detailed in the thesis document. These results should lead to exploitation of the code as library by the Kontron company.

Table des matières

Résumé **i**

Abstract **ii**

Abréviations **x**

1 Introduction **1**
- 1.1 Contexte et objectifs 1
- 1.2 Contributions 2
- 1.3 Organisation 3

2 Applications visées et modélisation **4**
- *Motivations* 4
- 2.1 Transformée de Fourier discrète et algorithme FFT de base 4
  - 2.1.1 Transformée de Fourier discrète 4
  - 2.1.2 Version de base FFT de Cooley-Tukey 6
  - 2.1.3 Calcul optimisé du bloc de base « papillon » 8
  - 2.1.4 Opération bit reverse 12
- 2.2 Variantes algorithmiques de la FFT 13
  - 2.2.1 Cooley-Tukey 14
  - 2.2.2 Radix-2 DIF 14
  - 2.2.3 Mixed radix 15
  - 2.2.4 Split-Radix FFT 16
  - 2.2.5 Stockham 16
  - 2.2.6 Autres versions 18
  - 2.2.7 Analyse et comparaison entre les versions 18
- 2.3 Applications basées sur la FFT : exemple de détection Radar 18
- *Bilan et discussion* 21

3 Les architectures de calcul intensif **22**
- *Motivations* 22
- 3.1 Parallélisme "on-chip" (généralités) 22
  - 3.1.1 Parallélisme d'instructions 22
    - 3.1.1.1 FMA *fused multiply-add* 23
    - 3.1.1.2 SMT *simultaneous multithreading* 23
  - 3.1.2 Le multicœur 25
- 3.2 Modèle SIMD pour CPU 26

3.2.1 Principes et modèle de programmation 26
3.2.2 Mise en œuvre (intrinsics) 27
3.2.3 Permutations de valeurs 29
3.3 Modèle SIMT pour GPU 30
3.3.1 Évolution des GPU 30
3.3.2 Principes et modèle de programmation 31
3.3.3 Lien entre GPU et CPU, et transferts de données 32
3.3.4 Mise en œuvre (OpenCL) 34
3.4 Architecture *Intel Core* considérée chez Kontron 36
3.4.1 GPU Intel 36
3.4.2 Unité d'exécution 38
3.4.3 Hiérarchie mémoire 40
3.4.4 Interconnexion 41
3.5 Mesures expérimentales sur les architectures *Intel Core* 42
3.5.1 Débit de calcul maximum 42
3.5.2 Taille des *work-groups* 44
3.5.2.1 Débit de transfert mémoire et *work-groups* 45
3.5.2.2 Calcul et taille du *work-group* 46
Bilan et discussion 47

4 Adaptation de la FFT sur l'architecture 49
Motivations 49
4.1 Adaptation SIMD puis SIMT sur un bloc élémentaire GPU 50
4.1.1 Démarche d'adaptation sur GPU 50
4.1.2 Cœur de calcul 51
4.1.3 Approche locale (*« in-register »*) 52
4.1.3.1 Taille maximale de la FFT 53
4.1.3.2 Optimisation de l'espace registres 54
4.1.3.3 SIMD sur GPU en OpenCL 55
4.1.4 Approche SIMT 57
4.1.4.1 Adaptation au registres 57
4.1.4.2 Adaptation à la mémoire partagée 59
4.1.4.3 Organisation des calculs FFT 62
4.1.4.4 Approche adaptative FFT 65
4.1.5 Synthèse de code 68
4.1.6 Expérimentation et résultats pratiques 68
4.2 Adaptation SIMD sur un cœur CPU 70
4.2.1 Cœur de calcul 71
4.2.2 Première implémentation SIMD 72
4.2.3 Deuxième implémentation SIMD : Adaptation des permutations 76
4.2.4 Expérimentation et résultats pratiques 79
Bilan et discussion 81

5 Performances système et distribution de l'application globale 82
Motivations 82
5.1 Charges utiles de calculs FFT 82

5.1.1 Sur multicœur CPU 83
5.1.1.1 Performances CPU 83
5.1.1.2 Dissipation thermique et consommation 84
5.1.2 Sur GPU 85
5.1.3 Conséquences 88
5.2 Dimensionnement pour application radar 88
5.2.1 Besoins en calculs FFTs 88
5.2.2 Adaptation de la charge de calcul aux contraintes 89
Bilan et discussion 91

6 Conclusion et Travaux Futurs **92**
6.1 Related Works 92
6.2 Développements futurs et perspectives 93

Table des figures

2.1 Décomposition d'un signal 5
2.2 Évolution des *twiddle factors* dans le temps 6
2.3 *Signal flow graph* FFT basique 9
2.4 Bloc de base papillon FFT 9
2.5 Opérations papillon standard 9
2.6 Opérations papillon optimisé 11
2.7 Précision de calcul FFT 12
2.8 Bit reverse 8 points 13
2.9 Convolution et Bit reverse 13
2.10 SFG FFT DIT 8 14
2.11 SFG FFT DIF 8 15
2.12 Mixed radix FFT 8 points 15
2.13 Split-radix FFT 17
2.14 Signal flow graph Algorithme de Stockham 17
2.15 Choix matériels pour un système Radar 19
2.16 Schéma bloc RDA SAR 20

3.1 Flux d'instruction classique 24
3.2 Avec hyperthreading 24
3.3 Impact de l'hyperthreading sur la FFT 25
3.4 Schéma CPU multicœur 26
3.5 Addition vectorielle SIMD de deux vecteurs 27
3.6 Évolution des unités SIMD 28
3.7 Benchmark FFT sur SIMD CPU 28
3.8 Instruction de permutation SIMD 30
3.9 Architecture d'un Execution unit GPU 31
3.10 Organisation des unités d'exécution du GPU d'AMD E6760 31
3.11 Exécution SIMT en mode cohérent 32
3.12 Exécution SIMT en mode divergent 32
3.13 SIMT Taxonomie 32
3.14 interconnexion CPU-GPU 33
3.15 débits PCIe 33
3.16 Architecture CPU Intel IvyBridge 33
3.17 Exemple *kernel* OpenCL qui additionne deux vecteurs d'entiers 34
3.18 Type vectoriel OpenCL 35
3.19 Compilation OpenCL 36
3.20 Architecture GPU Intel IvyBridge 37

3.21 Mapping sur EU GPU intel 38
3.22 Architecture Mémoire GPU Intel 40
3.23 Interconnexion CPU GPU intégrés Intel 41
3.24 Benchmark *Kernel* OpenCL GFlops maximum 43
3.25 performances crête 44
3.26 Passage à l'échelle des performances 44
3.27 Benchmark *kernel* OpenCL qui transfère les données de la mémoire DRAM vers les registres 45
3.28 Mise à l'échelle des débit Registres DRAM Work-group size =1 46
3.29 Mise à l'échelle des débit Registres DRAM Work-group size =2 46
3.30 Mise à l'échelle des débit Registres DRAM Work-group size =4 46
3.31 Mise à l'échelle des débit Registres DRAM Work-group size =8 46
3.32 Mise à l'échelle des débit Registres DRAM Work-group size =16 47
3.33 Mise à l'échelle des débit Registres DRAM Work-group size =32 47
3.34 Taille du work-group 47

4.1 Paramètres d'adaptation FFT 51
4.2 Adaptation FFT sur GPU niveau registres 54
4.3 Scalairisation 56
4.4 Mapping Radix-2 sur 1 *work-item* 57
4.5 Mapping FFT radix-4 sur 1 thread logique ou *work-item* 59
4.6 Mapping FFT radix-8 sur 1 thread logique ou *work-item* 59
4.7 Utilisation de la mémoire partagée SLM pour échanger 8KB de données FFT 1024 points complexes 60
4.8 Échange de données implémentation FFT 1K (radix-2) 61
4.9 Organisation des calculs FFT sur un *work-item* 63
4.10 Mouvement des données pour une FFT 1K points sur GPU 63
4.11 Organisation de l'implémentation FFT 1K sur un *work-item* 64
4.12 Organisation de l'implémentation FFT 1K 64
4.13 Design Space FFT 66
4.14 Radix-2 66
4.15 Radix-4 66
4.16 Radix-8 66
4.17 Performances FFT 1024 sur GPU IvyBridge et Haswell 69
4.18 Efficacité de notre implémentation FFT 1024 sur GPU IvyBridge et Haswell 70
4.19 Optimisation papillon FFT entrelacé 72
4.20 Simdization FFT AVX 73
4.21 Vectorisation AVX de la FFT 73
4.22 Comparaison performances FFT CPU sur Haswell 74
4.23 Performances par passes FFT sur CPU 75
4.24 Architecture d'un cœur CPU Haswell d'Intel 76
4.25 Organisation FFT vue globale 77
4.26 Quatre dernières passes FFT 78
4.27 Quatre dernières passes FFT choix 1 78
4.28 Quatre dernières passes FFT choix 2 79

4.29 Performances par passes FFT sur CPU version 1 80
4.30 Performances par passes FFT sur CPU version 2 80
4.31 Performances FFT sur CPU Comparatif 81

5.1 Passage à l'échelle FFT sur CPU 83
5.2 Effet d'emballement thermique 84
5.3 Expérience de placement des charges de calcul selon la localité du cœurs 85
5.4 Résultats thermiques sur le CPU IvyBridge d'Intel : 5°C de gain entre les cas favorables (C0,C3 et C4) et les cas défavorables (C1,C2 et C5) . . 85
5.5 Courbe des performances IVB 86
5.6 Courbe des performances HSW 86
5.7 analyse structure de notre kernel FFT 87
5.8 Tableau de correspondance débit mémoire et performances FFT 87
5.9 Distribution des calculs FFT sur CPU et GPU (sur *Intel IvyBridge*) . . . 89
5.10 Besoin de l'application en puissance de calcul afin d'avoir du temps réel (1555 fois /seconde) 90
5.11 Architecture de système de calcul radar à base de cartes CPU *Intel Haswell* 91

Liste des tableaux

2.1 Classification transformées de Fourier 5

3.1 Comparaison terminologie CUDA , OpenCL et OpenCL Intel 35

4.1 Taille du design space pour la FFT 67
4.2 Intel Ivy Bridge GPU benchmarks subset 67
4.3 Intel Haswell GPU benchmarks subset 67

Abréviations

Terme	Description
APU	Accelerated Processing Unit (*AMD*)
ARF	Architecture Register File
AVX	Advanced Vector Extensions
CS	commande streamer
ECC	Error Correction Code
EU	Execution Unit
FMA	Fused Multiply-Add
FPU	Floating-point unit
GRF	General purpose Register File
GTI	Graphics Technology Interface
HSW	Haswell (*Intel*)
ISA	Instruction Set Architecture
IVB	IvyBridge (*Intel*)
LLC	Last Level Cache
MIMD	Multiple instruction multiple data
MMU	Memory Management Unit
SIMD	Single instruction multiple data
SIMT	Single instruction multiple threads
SLM	Shared local memory
SMT	Simultaneous Multi-Threading
SPMD	Single program multiple data
SMT	Simultaneous Multi-Threading
SWaP	Size Weight and Power
TDP	Thermal Design Power
VLIW	Very long instruction word

Chapitre 1

Introduction

La prolifération du parallélisme au sein des calculateurs est une tendance de fond maintenant bien établie. Les supercalculateurs des années 80s sont complémentés par les grappes/clusters de calcul, les data centers du *cloud computing* et autres *network processors* ; les ordinateurs individuels sont désormais multi-cœurs et utilisent des co-processeurs graphiques dédiés (GPU) ; les téléphones mobiles et autres processeurs embarqués contiennent de multiples accélérateurs graphiques de traitement du signal (DSP)... Désormais, nous assistons même au mélange et aux interactions entre ces formes multiples de parallélisme au niveau physique : les GPU apparaissent tant dans des supercalculateurs du top 500 que dans des plates-formes embarquées, la concurrence entre architectures MPPAs, FPGA ou GPU fait l'objet de discussions et débats au sein de la communauté...

La recherche de l'efficacité (en performance ou en consommation) atteint son paroxysme. Néanmoins, elle se traduit trop souvent par une perte de l'automatisation lors des phases de compilation, quand le parallélisme (*concurrency*) inhérent potentiel des applications doit être projeté et ajusté au parallélisme (*parallelism*) réel des architectures et infrastructures matérielles sous-jacentes. Des armées d'ingénieurs réimplantent et portent souvent des programmes applicatifs manuellement d'une machine à sa remplaçante, ce qui induit des pertes en productivité et en sûreté du design à la compilation qui viennent souvent compenser les gains en efficacité à l'exécution.

1.1 Contexte et objectifs

Dans le cadre concret de cette thèse, financée par un contrat CIFRE entre Inria et la société Kontron, nous nous intéressons spécifiquement à l'architecture *Intel Core* (processeurs grand public IvyBridge, Haswell puis Broadwell), qui allie sur une même puce une architecture quadri-cœur CPU (x86) avec des ALU vectorielles SIMD, et des cœurs de calcul GPU avec une programmation de nature SIMT (*Single Instruction Multiple Threads*). La société Kontron assemble des sous-systèmes sur cartes-mères à partir de tels composants, pour des applications de nature professionnelle qui réclament à la fois efficacité et robustesse. Dans l'exemple dont nous traiterons (de manière simplifiée) au cours de cette thèse, une application radar nécessite suffisamment de puissance de calcul afin de calculer des opérations de traitement de signal

(transformée de Fourier rapide principalement) en parallèle, en nombre important et en temps-réel. Notre approche du problème (telle qu'elle nous est progressivement apparue au cours de nos travaux) consistera à optimiser le traitement d'une FFT (unique) sur un cœur (unique) de calcul, qu'il soit CPU ou GPU. Cette approche a comme avantage de garantir la localité des données lors de ce module de calcul, alors que les transferts de données (avec les problèmes de caches et de débit des bus internes d'interconnexion) peuvent se révéler un aspect excessivement pénalisant de la parallélisation. Ensuite, on devra dimensionner, entre les (4) cœurs CPU du processeur et les (16 pour Ivybridge puis 40 pour Haswell) cœurs GPU d'une même carte, afin de répartir au mieux les nombreuses occurrences de calcul FFT élémentaires, et au-delà dimensionner le système afin d'avoir la capacité de calcul désirée en utilisant plusieurs processeurs *Intel Core* sur une même carte.

1.2 Contributions

Le placement optimisé d'une FFT est donc un point central de notre approche. Il doit être effectué (une bonne fois pour toute, sur un cœur CPU et sur un cœur GPU) de manière spécifique, du fait que la définition récursive de cet algorithme ne suit pas les schémas de base pour l'automatisation générique de la compilation parallèle (par les méthodes polyédriques notamment). Ensuite, la répartition des FFT entres les différentes ressources et cœurs de calcul ressemblera (de manière caricaturale) à une sorte de gigantesque « règle de 3 » pour équilibrer les charges de calcul. D'un point de vue pratique (et sensible pour la société Kontron), cette phase a aussi l'avantage de produire un code source natif, compréhensif et modifiable, alors que dans la situation actuelle les bibliothèque natives Intel pour le calcul FFT sur CPU ne fournissent qu'un code cible moyennement optimisé, et qu'il n'en existe simplement pas pour la compilation GPU. Nous avons également considéré les initiatives communautaires qui proposent des études et solutions optimisées (FFTW, Spiral,...)[1], mais elles ne répondent pas complètement à nos besoins en terme de « *fine tuning* » sur notre architecture, ou de disponibilité de code source pour exploitation commerciale. L'objectif ultime de nos travaux sera donc, non seulement de produire un code source efficace optimisé et dimensionné pour nos architectures cibles, mais encore d'étudier comment synthétiser ce code de manière modulaire à partir de certains paramètre dimensionnants simples. On pourra alors espérer produire de manière automatique les versions correspondant aux versions ultérieures des processeurs *Intel Core*, qui semblent devoir garder des architectures similaires tout en augmentant les tailles des paramètres et en multipliant les nombres de cœurs.

Les paramètres qui entrerons en ligne de compte dans nos calculs seront principalement le nombre de points requis pour la taille de FFT : 1014 généralement, ce qui correspond à 10 étages successifs de calculs de papillons élémentaires ($1014 = 2^{10}$) ; ce nombre d'étages est en fait le paramètre direct qui nous concerne. Le second paramètre d'importance sera le nombre de registres internes (ou mémoire L0) disponible dans chaque cœur (ou dans chaque bloc élémentaire de calcul) ; avec ces nombres on peut déterminer combien d'étages peuvent être exécutés sans de mouvements de données externes (juste dans les registres), ce qui fournit une information essentielle sur la granularité du découpage de la FFT, dont la réécriture à partir de blocs amal-

gamés comportant le bon nombre d'étages permet d'obtenir le code final. Tous ces points forment l'essentiel des contributions techniques de ce document.

1.3 Organisation

Le document est structuré en 6 parties, comme suit : après cette introduction, le chapitre 2 introduit la problématique du calcul de la Transformée de Fourier Discrète, avec toutes ses variantes et optimisation connues ; le chapitre décrit également la modélisation abstraite d'une application radar qui utilise intensivement cette transformée. Le chapitre 3 décrit les éléments majeurs d'architecture et de micro architecture parallèle, jusqu'à les spécialiser pour notre architecture *Intel Core* (qui allie cœurs CPU SIMD X86 et cœurs GPU SIMT embarqués sur une même puce). Le chapitre 4 contient l'essentiel de nos expérimentations et résultats ayant donné lieu à la solution de placement (une FFT sur un cœur) que nous avons définie et mise en œuvre. Il contient des résultats expérimentaux validant cette approche. À noter que certains de nos résultats ou définitions « ajustées » pour préparer ces résultats se trouvaient déjà présentées dans les chapitres précédents. Le chapitre 5 décrit le placement plus général des calculs de l'application radar sur l'intégralité des ressources d'une carte à base de processeurs *Intel Core*. Pour des raisons de temps nous n'avons pas encore pu explorer ici toutes les pistes d'optimisation et d'automatisation, mais nous savons décrire assez précisément les travaux effectués, et ceux envisagés. Enfin le chapitre 6 apporte une conclusion au travail en traçant la piste de développements futurs, ainsi que les enseignements qu'on peut en tirer dès maintenant.

Chapitre 2

Applications visées et modélisation

Motivations

Les applications RADAR et SONAR sont deux domaines exigeants en puissance de calcul. Plus particulièrement, pour les traitements RADAR les latences de calcul doivent être réduites au minimum sous peine de ne pas pouvoir détecter en temps réel les cibles en mouvement. Ces applications font appel à des bibliothèques mathématiques contenant une liste exhaustive de fonctions élémentaires telles que les transformées, les filtres et les opérations matricielles. Ce sont des algorithmes de type flot de données opérant sur de larges tableaux d'une ou plusieurs dimensions. Parmi les différents algorithmes qui composent ces bibliothèques, il en est un qui revêt une importance capitale dans la plupart des cas d'utilisation. Il s'agit de la transformée de Fourier rapide.

L'optimisation de la transformée de Fourier sur des architectures parallèles est donc primordiale pour atteindre des performances temps réels dans les systèmes Radar : l'effort d'optimisation sur cette brique de base se traduit immédiatement par un gain majeur de performance sur tout le système.

Le but de ce chapitre est de présenter les fondamentaux de cette transformée et de son application, en portant un éclairage particulier sur les aspects qui nous intéresserons spécialement par la suite, comme les subtiles variations algorithmiques qui permettront un ajustement à une architecture spécifique donnée. Nous conclurons le chapitre par une contribution technique, qui nous semble nouvelle, réorganisant le calcul à partir des coefficients, et qui en pratique améliore la précision du résultat.

2.1 Transformée de Fourier discrète et algorithme FFT de base

2.1.1 Transformée de Fourier discrète

Les processus physiques peuvent être décrits dans le domaine temporel à l'aide de la valeur d'une quantité h en fonction du temps t, ou bien dans le domaine fréquentiel à l'aide de son amplitude H en fonction de sa fréquence f. On peut alors considérer que $h(t)$ et $H(f)$ sont deux représentations de la même fonction modulo une transformation. La transformée de Fourier permet le passage d'une représentation à une autre.

Selon le type des signaux à transformer, différents types de transformées de Fou-

rier sont définis :

	Non périodique	Périodique
Continue	Transformée de Fourier	Série de Fourier
Discret	DTFT (discrete time Fourier transform) [1]	DFT (Discrete Fourier transform)

TABLE 2.1 – Classification transformées de Fourier

Pour un signal discret et périodique la transformée correspondante porte le nom de transformée de Fourier discrète (on utilise habituellement l'abréviation anglaise DFT pour "Discrete Fourier Transform"). Nous nous focaliserons exclusivement sur ce type de transformée dans ce document.

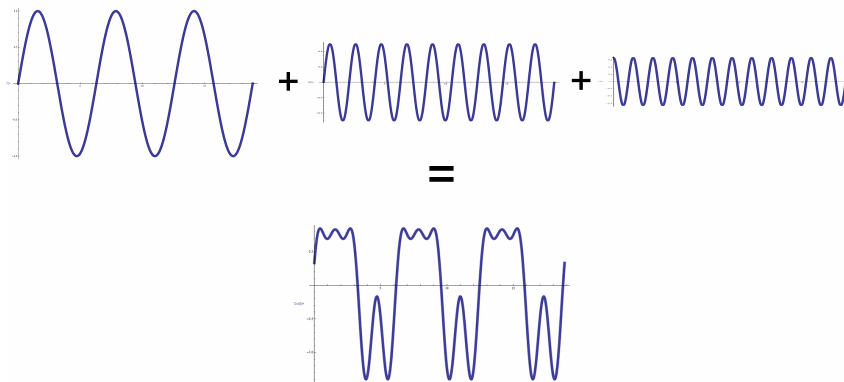


FIGURE 2.1 – Décomposition d'un signal en plusieurs signaux sinusoïdaux

Soit une séquence d'entrée $x(n)$, la DFT de N points est définie comme suit :

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk} \qquad \textbf{avec :} \, k = 0, 1, \dots, N-1 \tag{2.1}$$

Où l'entier n est l'index de temps, l'entier k est l'index de fréquence et le nombre complexe W_N^{nk} qui correspond à la racine n-ième de l'unité, communément appelé *twiddle factor*, est définie comme suit :

$$W_N^{nk} = \exp\left(\frac{-2i\pi nk}{N}\right) = \cos\left(\frac{2\pi nk}{N}\right) - i \cdot \sin\left(\frac{2\pi nk}{N}\right). \tag{2.2}$$

La DFT inverse (IDFT) est exprimé comme ceci :

$$x(n) = \frac{1}{N} \cdot \sum_{n=0}^{N-1} X(n) \cdot W_N^{-nk} \qquad \textbf{avec :} \, k = 0, 1, \dots, N-1 \tag{2.3}$$

Nous observons que N multiplications complexes et $N-1$ additions complexes

1. Ces signaux sont définis que dans un espace temporel discret, cette transformée est discrète dans l'espace temporel et continue dans l'espace fréquentiel.

sont nécessaires pour calculer un point, donc il nous faut N^2 multiplications complexes et $N^2 - N$ additions complexes pour calculer une DFT/IDFT de N échantillons.

Le calcul direct de la DFT est inefficace avec l'augmentation de la taille du signal à transformer.

2.1.2 Version de base FFT de Cooley-Tukey

La transformée de Fourier rapide a été « décrétée » l'un des 10 algorithmes majeurs du 20e siècle [2]. C'est aussi l'un des plus utilisés et en 1990 il a été estimé [3] que sur une base installée de supercalculateurs Cray de 200 machines (à 25 millions de dollar US l'unité), 40% des cycles CPU sont dédiés aux calculs de la FFT.

L'algorithme de la transformée de Fourier rapide a été initialement découvert par Gauss en 1805, mais il n'a eu de succès qu'en 1965 après la publication [4] de celui-ci par J. W. Cooley et J. W. Tukey. C'est pour cette raison que l'algorithme de base porte habituellement leurs noms.

L'algorithme de la FFT est un algorithme de type « Diviser pour régner », il factorise la DFT pour réduire le nombre d'opérations de $O(N^2)$ à $O(N \cdot logN)$.

Nous pouvons visualiser la structure récursive de la FFT à travers la figure 2.2 qui montre comment les racines de l'unité sont distribuées tout au long de l'algorithme de la FFT.

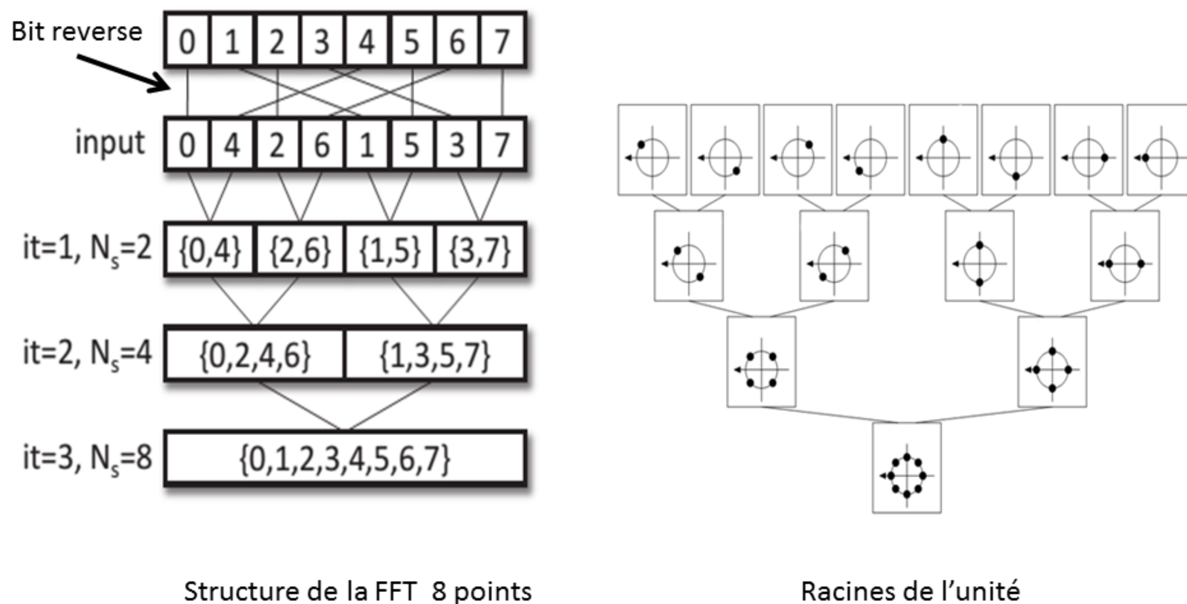


FIGURE 2.2 – Évolution de l'utilisation des *twiddle factors* à travers les passes (FFT 8 points)

Nous verrons plus bas que la structure récursive est néanmoins généralement altérée par le besoin de réordonner les données entre deux passes élémentaires de calcul, l'opération intermédiaire est appelée *bit reverse*.

On verra ultérieurement des variantes optimisant plus encore la version de base, proposant de nouveaux schémas récursifs préservant la complexité globale des calculs.

L'algorithme de la FFT décrit par Cooley et Tukey [4] dans leur article se dérive

alors comme ceci :

Soit la DFT de $x(n)$:

$$X(k) = \sum_{n=0}^{N-1} x(n).W_N^{nk} \qquad \textbf{avec :} \; W = e^{\frac{2\pi i}{N}} \tag{2.4}$$

Supposons que N s'écrit sous la forme de : $N = r_1 \times r_2$

Soit k et n définit par :

$$k = k_1 r_1 + k_0 \qquad \textbf{avec :} \; k_0 = 0, 1, \ldots, r_1 - 1 \;\; \textbf{et} \;\; k_1 = 0, 1, \ldots, r_2 - 1 \tag{2.5}$$

$$n = n_1 r_2 + n_0 \qquad \textbf{avec :} \; n_0 = 0, 1, \ldots, r_2 - 1 \;\; \textbf{et} \;\; n_1 = 0, 1, \ldots, r_1 - 1 \tag{2.6}$$

$$X(k_1, k_0) = \sum_{n_0=0}^{r_2-1} \sum_{n_1=0}^{r_1-1} x(n_1, n_0) \cdot W^{kn_1 r_2} W^{kn_0} \tag{2.7}$$

avec : $W^{kn_1r_2} = W^{k_0n_1r_2}$

$$X(k_1, k_0) = \sum_{n_0=0}^{r_2-1} \sum_{n_1=0}^{r_1-1} x(n_1, n_0) \cdot W^{k_0 n_1 r_2} W^{kn_0} \tag{2.8}$$

On pose :

$$X_1(k_0, n_0) = \sum_{n_1=0}^{r_1-1} x(n_1, n_0) \cdot W^{k_0 n_1 r_2} \tag{2.9}$$

$$X(k_1, k_0) = \sum_{n_0=0}^{r_2-1} X_1(k_0, n_0) \cdot W^{(k_1 r_1 + k_0) n_0} \tag{2.10}$$

Le tableau X_1 contient N éléments, chacun nécessite r_1 opérations, ce qui donne un total de $N \cdot r_1$ opérations pour obtenir X_1. Pour calculer X à partir de X_1 il faut $N \cdot r_2$ opérations. Donc, pour cet algorithme qui décompose le calcul en deux étapes, nous avons besoin au total de : $N(r_1 + r_2)$ opérations ce qui est largement inférieur au nombre d'opérations initial qui était de N^2 opérations.

Si on répète cette même décomposition successivement nous aurons un algorithme avec m étapes et $N(r_1 + r_2 + ... + r_m)$ opérations avec $N = r_1 r_2 ... r_m$. Ce qui est toujours inférieur ou égal à N^2.

Donc si $r_1 = r_2 = \ldots = r_m = r$ alors $m = \log_r N$ et le nombre total d'opérations devient :

$$rN \log_r N \tag{2.11}$$

Si $r = 2$ alors $N = 2^m$

Les indices n et k s'écrivent alors sous la forme :

$$k = k_{m-1} 2^{m-1} + ... + k_1.2 + k_0 \tag{2.12}$$

$$n = n_{m-1}2^{m-1} + \ldots + n_1.2 + n_0 \quad (2.13)$$

Avec n_x et k_x prenant comme valeurs soit 0 soit 1, ce qui représente les valeurs des bits dans la représentation binaire de k et de n.

Alors la transformée de Fourier s'écrit en fonction de la position binaire des indices :

$$X(k_{m-1}, \ldots, k_0) = \sum_{n_0=0}^{1} \sum_{n_1=0}^{1} \cdots \sum_{n_{m-1}=0}^{1} x(n_{m-1}, \ldots, n_0) \cdot W^{kn_{m-1}2^{m-1}+\ldots+kn_0} \quad (2.14)$$

Avec : $W^{kn_{m-1}\cdot 2^{m-1}} = W^{k_0 n_{m-1}\cdot 2^{m-1}}$

Donc la somme interne d'indice n_{m-1} dans l'équation ne dépend que de $k_0, n_{m-2}, \ldots, n_0$ On l'écrit alors :

$$X_1(k_0, n_{m-2}, \ldots, n_0) = \sum_{n_{m-1}=0}^{1} x(n_{m-1}, \ldots, n_0) \cdot W^{k_0 n_{m-1}2^{m-1}} \quad (2.15)$$

On réitère cette même procédure aux autres sommes en utilisant :

$$W^{kn_{m-p}\cdot 2^{m-p}} = W^{(k_{p-1}2^{p-1}+\ldots+k_0)n_{m-p}\cdot 2^{m_p}} \quad (2.16)$$

On obtient pour $p = 1, 2, \ldots, m$

$$X_p(k_0, \ldots, k_{p-1}, n_{m-p-1}, \ldots, n_0) = \sum_{n_{m-p}=0}^{1} X_{p-1}(k_0, \ldots, k_{p-2}, n_{m-p}, \ldots, n_0)\cdot W^{(k_{p-1}2^{p-1}+\ldots+k_0)n_{m-p}\cdot 2^{m-p}} \quad (2.17)$$

La dernière somme donne le résultat final de la transformée de Fourier :

$$X(k_{m-1}, \ldots, k_0) = X_m(k_0, \ldots, k_{m-1}) \quad (2.18)$$

On remarque que l'index de X doit avoir son ordre binaire inversé pour que le résultat soit correct : c'est pour cela qu'une étape de *bit reversing* est nécessaire pour l'algorithme initial de la FFT publié par Cooley et Tukey. Cette étape sera décrite dans la section 2.1.4.

Nous notons aussi que le graphe de dépendance de données doit rester inchangé, cependant l'ordre des index peut varier tout au long de l'algorithme. Nous introduirons ici le graphe de flot de données utilisé dans la suite de cette thèse pour illustrer les différents algorithmes. Dans ce graphe, les données du même étage peuvent être arrangées librement en gardant les arêtes du graphe connectées. La figure 2.3 illustre le graphe de flot de données pour l'algorithme de Cooley et Tukey.

2.1.3 Calcul optimisé du bloc de base « papillon »

Tous les algorithmes FFT sont construits autour d'un bloc de base qui porte usuellement le nom de papillon (figure 2.4). Ce bloc prend deux valeurs complexes en en-

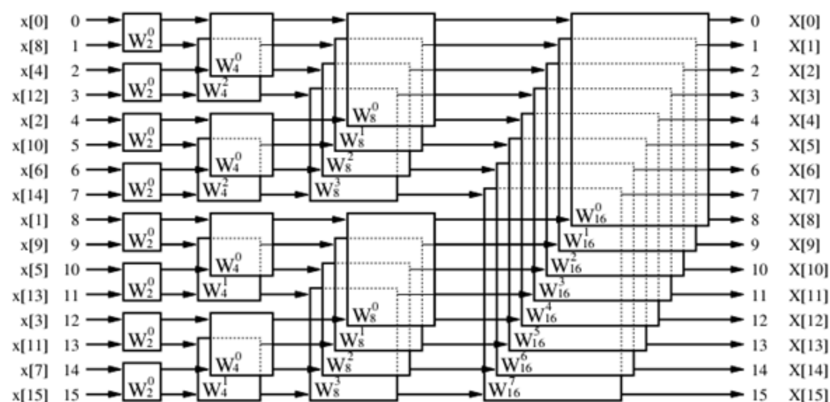


FIGURE 2.3 – *Signal flow graph* basique FFT de taille 16

trée et produit aussi deux valeurs complexes en sortie. Il nécessite la multiplication de deux nombres complexes. Cette opération est calculée à l'aide de 10 opérations flottantes (typiquement : 6 additions et 4 multiplications(figure 2.5)).

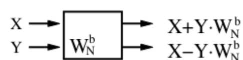


FIGURE 2.4 – Bloc de base papillon FFT

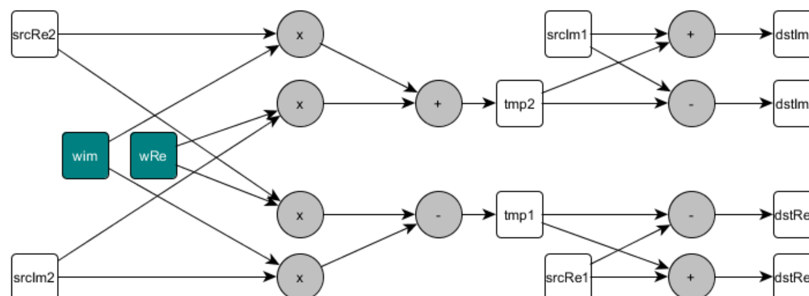


FIGURE 2.5 – Opérations papillon standard

Une instruction ad hoc permettant d'enchainer une multiplication et une addition $a = a + b \times c$ en une seule opération élémentaire, existe dans la plupart des architectures matérielles modernes (à partir des processeurs DSP). On la nomme généralement MAC (*multiply-accumulate*) pour le calcul en précision fixe, et elle porte aussi le nom de FMA (*fused multiply-add*)[5] pour les calculs en nombres flottants. Dans cette thèse nous nous intéressons aux calculs flottants.

Cette instruction apparait désormais dans les jeux d'instructions des GPU qui font partie de nos architectures cibles.

Nous souhaitons alors utiliser efficacement l'instruction FMA, qui permet de faire l'addition et la multiplication dans le même temps qui serait nécessaire pour faire soit une addition soit une multiplication.

En 1993 Linzer et Feig[6] [7] sont les premiers à exploiter la FMA dans les algorithmes de la FFT radix-2, radix-4 et split-radix. En 1997 Goedecker[8] a proposé un

algorithme qui exploite la FMA moins complexe que les algorithmes de Linzer et Feig.

Aussi dans les travaux d'une thèse antérieure à celle-ci [9], portant sur les architectures PowerPC à Kontron, l'utilisation de l'instruction FMA a été exploitée à travers une factorisation des calculs afin d'exprimer toutes les opérations en une suite de FMA (6 au total). Ce nombre représente l'optimal en nombre d'instructions FMA, car il n'est pas possible de calculer un papillon FFT avec moins de 6 FMA.

La factorisation effectuée pour exploiter la FMA dans les travaux antérieurs à cette thèse est de type :

$$ax + by \to a(x + (b/a)y) \tag{2.19}$$

Nous notons a_r et b_r la partie réelle des valeurs d'entrée de notre papillon et a_i et b_i leurs parties imaginaires. De même ω_r et ω_i les parties réelles et imaginaires du *twiddle factor*. Les résultats de notre papillon sont notés A_r et B_r pour les parties réelles, A_i et B_i pour les parties imaginaires. Le papillon standard est alors calculé en 10 opérations selon les équations suivantes (2.20),(2.21) :

$$A_r = a_r + (\omega_r \cdot b_r - \omega_i \cdot b_i) \qquad A_i = a_i + (\omega_i \cdot b_r + \omega_r \cdot b_i) \tag{2.20}$$

$$B_r = a_r - (\omega_r \cdot b_r - \omega_i \cdot b_i) \qquad B_i = a_i - (\omega_i \cdot b_r + \omega_r \cdot b_i) \tag{2.21}$$

En appliquant la factorisation décrite plus haut, le papillon devient :

$$A_r = a_r - (b_i - \frac{\omega_r}{\omega_i} \cdot b_r) \cdot \omega_i \qquad A_i = a_i + (\frac{\omega_r}{\omega_i} \cdot b_i + b_r) \cdot \omega_i \tag{2.22}$$

$$B_r = a_r + (b_i - \frac{\omega_r}{\omega_i} \cdot b_r) \cdot \omega_i \qquad B_i = a_i - (\frac{\omega_r}{\omega_i} \cdot b_i + b_r) \cdot \omega_i \tag{2.23}$$

Cette factorisation est correcte dans le cas où $\omega_i \neq 0$, cependant quand $\omega_i = 0$ nous devons le remplacer par une valeur très petite, à savoir $\omega_i = 0,0000001$.

Cette méthode est très efficace mais elle présente un inconvénient du point de vue mathématique, car si nos *twiddle factors* sont remplacés par une valeur très petite dans le cas où ω_i est égal à zéro, ceci engendre une dégradation de la précision de calcul, cette dernière est mesurée plus tard.

Notre contribution [10] que nous décrivons ici permet de s'affranchir de cette limitation, et permet ainsi de bénéficier de la précision de calcul de l'instruction FMA. Nous savons que l'instruction FMA permet d'optimiser le calcul au niveau performances et aussi au niveau précision par rapport à une multiplication suivie d'une addition, car l'instruction FMA n'effectue l'arrondi qu'à la fin du calcul.

Notre factorisation s'écrit alors sous la forme suivante :

$$A_r = a_r + (b_r + \frac{-\omega_i}{\omega_r} \cdot b_i) \cdot \omega_r \qquad A_i = a_i + (\frac{\omega_i}{\omega_r} \cdot b_r + b_i) \cdot \omega_r \tag{2.24}$$

$$B_r = a_r + (b_r + \frac{-\omega_i}{\omega_r} \cdot b_i) \cdot (-\omega_r) \qquad B_i = a_i + (\frac{\omega_i}{\omega_r} \cdot b_r + b_i) \cdot (-\omega_r) \tag{2.25}$$

Nous avons ainsi factorisé notre calcul à l'aide de la partie réelle des *twiddle factors* ω_r. La particularité de cette factorisation est que ω_r ne seras jamais nul, ceci re-

vient au fait que $\tan(\frac{\pi}{2})$ n'est pas défini. Nous rappelons aussi que $\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}$ et que $\cos(\frac{\pi}{2}) = 0$.

Nous utiliserons cette contribution comme brique de base pour la suite de notre thèse (voir figure 2.6).

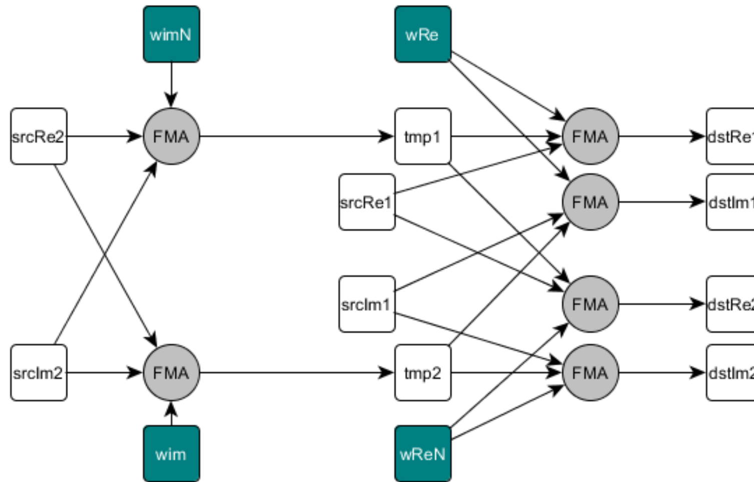


FIGURE 2.6 – Opérations papillon optimisé

Nous souhaitons aussi évaluer l'impact de cette factorisation sur la précision des calculs. Donc, afin de mesurer la précision des calculs de notre optimisation, plusieurs procédures existent. Pour des fins de reproductibilité est aussi pour pouvoir avoir des références communes avec l'état de l'art dans le domaine du calcul FFT, nous avons fait le choix d'adopter la méthode de vérification préconisée par FFTW [2] [1].

FFTW calcule la précision d'une FFT en comparant la transformation d'un signal pseudo-aléatoire uniforme compris entre -0.5 et $+0.5$ avec le résultat d'une FFT de référence dite exacte.

Le choix de la FFT de référence influe fortement sur le résultat, surtout que nous voulons comparer la précision de calculs d'algorithmes FFT entre elles. Nous devons alors nous affranchir de ce choix qui est à un certain niveau subjectif. Pour ce faire, nous avons choisi de comparer le résultat de la FFT suivie de la FFT inverse (signal reconstruit) avec le signal d'origine (signal source).

Pour comparer les deux signaux nous utilisons la fonction suivante $compare(FFTinv(FFT(signal_{Source})), signal_{Source})$ qui calcule la norme euclidienne (L2) relative définie dans l'équation 2.26.

$$compare(a, b) = \|a - b\|_2 / \|b\|_2 \tag{2.26}$$

Avec a, le résultat de notre FFT/FFT inverse, b le signal pseudo-aléatoire source dont les valeurs sont comprises dans l'intervalle $[-0.5, 0.5]$, et :

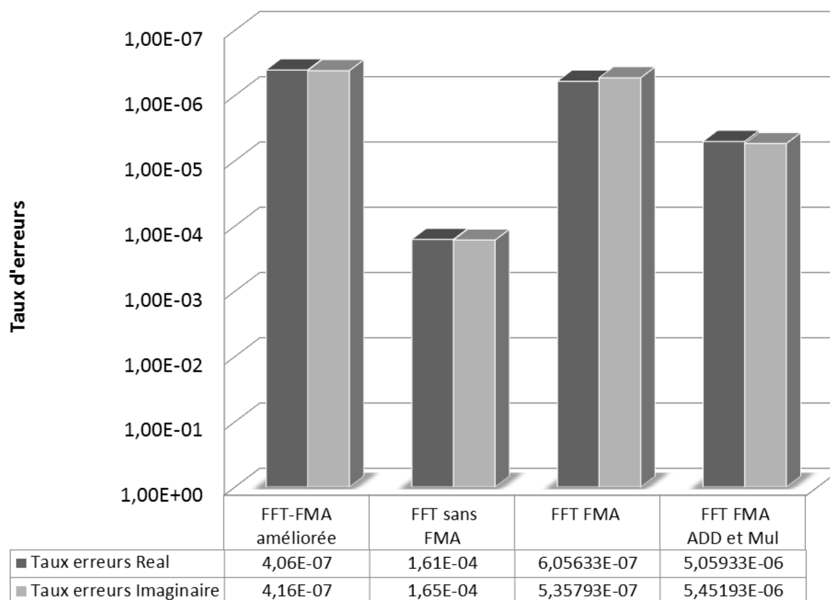
$$\|x\|_2 = \sqrt{\sum |x_i|^2} \tag{2.27}$$

Le graphique suivant (figure 2.7) montre le gain en précision obtenu en utilisant

2. http://www.fftw.org/accuracy/method.html

la FMA sur une FFT de 1024 points complexes sur le GPU d'Intel *IvyBridge HD4000* dont l'architecture sera étudiée finement dans cette thèse (chapitre 3).

Notons aussi que le taux d'erreurs accumulées entre une FFT et une FFT inverse est grandement amélioré par rapport à une implémentation qui n'utilise pas de FMA, surtout en simple précision sur GPU, en effet elle passe de 10^{-4} à 10^{-7}.



	FFT-FMA améliorée	FFT sans FMA	FFT FMA	FFT FMA ADD et Mul
Taux erreurs Real	4,06E-07	1,61E-04	6,05633E-07	5,05933E-06
Taux erreurs Imaginaire	4,16E-07	1,65E-04	5,35793E-07	5,45193E-06

FIGURE 2.7 – Précision de calcul FFT sur GPU Intel IvyBridge HD4000

2.1.4 Opération bit reverse

Le bit reverse se traduit par une organisation selon les bits du poids fort des données d'entrée initialement ordonnées selon le bit du poids faible. Ceci est du à la propriété de décompositions successives de la FFT en deux sous transformées.

Sequential order :	000,	001,	010,	011,	100,	101,	110,	111
	0,	1,	2,	3,	4,	5,	6,	7
Bit-reversed order :	000,	100,	010,	110,	001,	101,	011,	111
	0,	4,	2,	6,	1,	5,	3,	7

Il faut bien prendre en compte cette particularité car le ré-ordonnancement des données est très coûteux, surtout avec des accès en puissances de deux.

Pour certaines applications où nous avons besoin d'enchainer une FFT suivie par une FFT inverse, nous pouvons éviter l'étape du bit reverse. C'est notamment le cas pour la convolution.

Le bit reverse se fait à l'aide de permutations. On verra en section 3.2 que les architectures CPU modernes proposent dans certains cas des versions (limitées) de telles permutations.

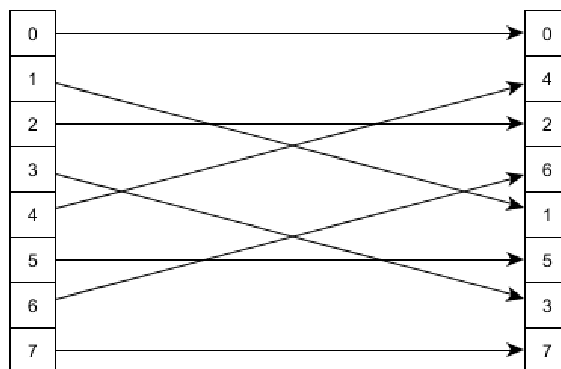


FIGURE 2.8 – Bit reverse de 8 points

La convolution : La convolution correspond à la réponse du filtre à une entrée donnée (notée $e(t)$). Le filtre est entièrement caractérisé par sa réponse impulsionnelle $h(t)$. Mise en équation, la réponse du filtre est $s(t) = h(t) * e(t)$.

Une particularité de la convolution dans le cas des filtres linéaires, si on entre un signal $e(t) = e^{2\pi i f t}$ le signal de sortie $s(t)$ sera aussi de la forme $e^{2\pi i f t}$ au facteur $H(f)$ près. Ce facteur n'est autre que la transformée de Fourier de $h(t)$.

C'est pour cette raison que la FFT est très pratique quand on souhaite faire du filtrage sur un signal, nous effectuons le filtrage dans l'espace fréquentiel en faisant une simple multiplication ponctuelle avec le filtre puis on enchaine avec une FFT inverse pour retrouver le signal filtré.

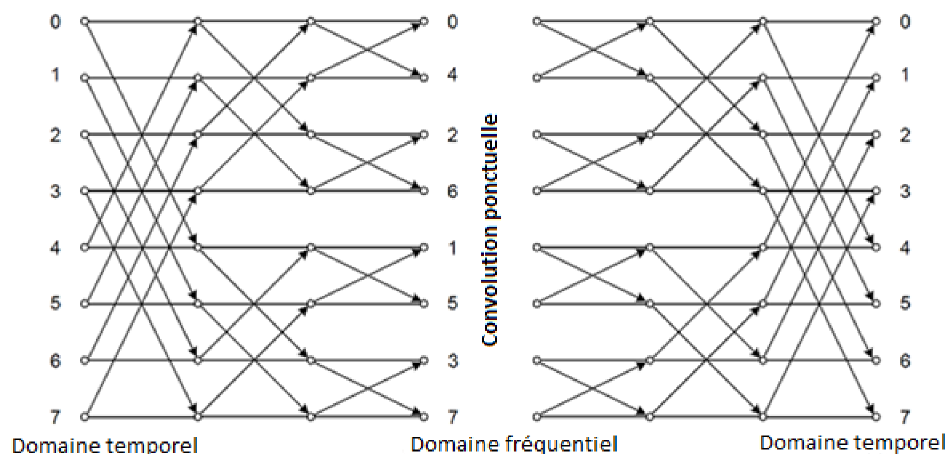


FIGURE 2.9 – Convolution FFT et Bit reversing

Pour une convolution l'étape du bit reverse n'est pas importante, on peut juste ignorer cette étape.

2.2 Variantes algorithmiques de la FFT

Au-delà de la version de base, il existe plusieurs méthodes pour regrouper et ordonner les calculs, ainsi que pour distribuer l'opération de *bit-reverse*. Si toutes ces

variantes conservent de fait le même graphe de dépendance pour le flot de données, elles consistent à encapsuler les calculs par des approches « *divide-and-conquer* » distinctes, et surtout à rapprocher les données afin d'augmenter la localité de l'application du bloc papillon, et de mieux répartir les opérations conduisant au bit reverse.

Elles ont donc la même complexité générale de $O(n \cdot logn)$ opérations, mais elles peuvent varier en efficacité pratique avec une mesure plus fine de cette complexité.

Les futures implantations parallèles considérées au chapitre 4, et leur ajustement à une architecture parallèle donnée comme décrite au chapitre 3, devront jouer sur ces variations afin de répartir les calculs en modules de base de taille appropriée. Nous décrivons maintenant les aspects majeurs des variantes algorithmiques dans l'absolu.

Pour plus de détails, le très bon livre de *Van Loan « Computational Frameworks for the Fast Fourier Transform »*[11] constitue un ouvrage de référence unificateur des différents algorithmes.

2.2.1 Cooley-Tukey

Cet algorithme est décrit dans la section 2.1.2 [4][12], il est l'implémentation populaire qui porte le nom radix-2 DIT (decimation in time) à entrelacement temporel.

Pour une FFT de N points en radix-2 DIT, les données d'entrée sont ordonnées en *bit-reversed order* et les sorties sont ordonnées en ordre naturel. Sachant qu'une transformée de Fourier directe permet le passage de l'espace temporel à l'espace fréquentiel, alors si l'étape de *bit reverse* est effectuée au début, on parle d'entrelacement temporel et si elle est faite à la fin, on parle d'entrelacement fréquentiel.

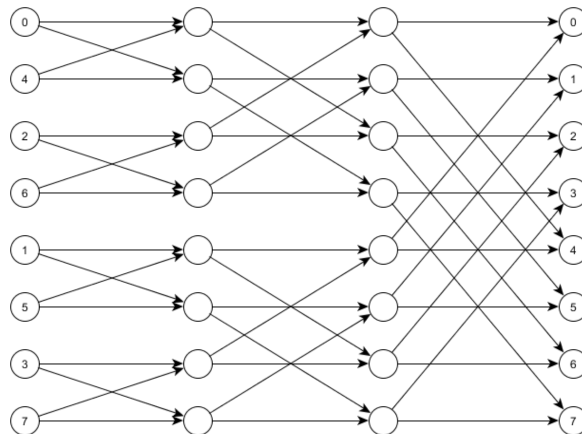


FIGURE 2.10 – Signal flow graph FFT décimation dans le temps (DIT de taille 8)

L'algorithme FFT radix-2 DIT (figure 2.10) est une succession de $log_2(N)$ étages, tous composés de $N/2$ papillons.

2.2.2 Radix-2 DIF

L'algorithme Radix-2 DIF (decimation in frequency) diffère de son prédécesseur par l'ordre des données d'entrées, ces dernières sont fournies dans l'ordre séquentiel et les sorties sont dans l'ordre *bit-reversed order*.

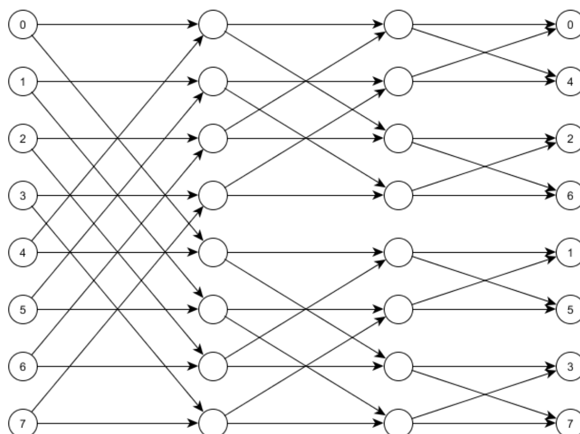


FIGURE 2.11 – Signal flow graph FFT décimation en fréquence (DIF de taille 8)

La complexité de calcul demeure inchangée.

2.2.3 Mixed radix

Une FFT peut être partitionnée en plusieurs sous transformées. Si toutes les partitions ont la même taille r, alors cette version de l'algorithme FFT est dite de type *radix-r* .

Dans le cas où les partitions ont des tailles différentes, on parle de *mixed-radix* FFT[13].

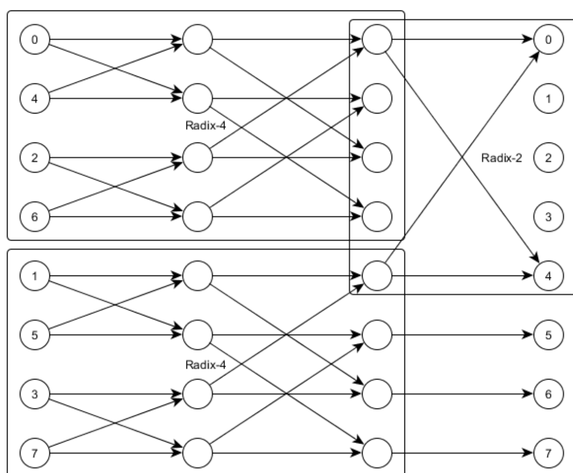


FIGURE 2.12 – Exemple de FFT 8 points avec l'algorithme *mixed-radix* : la première passe est calculée avec deux radix-4 et la dernière passe avec 4 radix-2

L'approche *mixed-radix* permet plus de dégrés de liberté dans l'implémentation et l'organisation des calculs. Dans cette thèse, une particulière importance est donnée à cette méthode.

2.2.4 Split-Radix FFT

L'algorithme FFT *split-radix* trouve son origine dans une observation très simple illustrée sur la figure 2.13 :

Le graphe d'un algorithme radix-2 à entrelacement temporel peut se transformer de manière évidente en graphe d'un algorithme radix-4 uniquement en changeant les exposants de la racine de l'unité servant de coefficients multiplicateurs (*twiddle factors*). Ce faisant, il apparait assez vite que, à chaque étage de l'algorithme, un radix-4 est plus intéressant pour les termes impairs, et un radix-2 pour les termes pairs de la FFT.

L'algorithme *split-radix* est donc basé sur la décomposition suivante :

$$X_k = \sum_{k=0}^{N-1} x_n W_N^{nk}, \tag{2.28}$$

Il divise récursivement le calcul de la FFT en une FFT de taille $N/2$ et deux FFT de taille $N/4$.

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n).e^{-(i.\frac{2\pi(2n)k}{N})} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1).e^{-(i.\frac{2\pi(4n+1)k}{N})} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3).e^{-(i.\frac{2\pi(4n+3)k}{N})} \tag{2.29}$$

Après identification des trois sous-transformées, il ressort l'écriture suivante :

$$X(k) = DFT_{\frac{N}{2}}[x(2n)] + w_N^k.DFT_{\frac{N}{4}}[x(4n+1)] + w_N^{3k}.DFT_{\frac{N}{4}}[x(4n+3)] \tag{2.30}$$

La combinaison des algorithmes radix-2 et radix-4 donne à son papillon, sa forme caractéristique en L.

Créé en 1984 par P. Duhamel et H. Hollmann [14], il représente l'algorithme possédant le plus faible nombre d'opérations flottantes, multiplications et additions confondues, sur des signaux de taille égale à une puissance de deux : pour un signal de taille n, le nombre d'opérations est $4n\log_2 n - 6n + 8$ Flop (avec $n = 2^m$).

Plus récemment, en 2007, ce nombre d'opérations arithmétiques nécessaire au calcul de l'algorithme a encore été amélioré par S.G. Johnson et M. Frigo [15]. Cet algorithme a aussi été amélioré par Bernstein [16], il a donné aussi le nom Tangente FFT à cet algorithme pour mettre en avant le rôle important que jouent les tangentes comme constantes dans cet algorithme.

2.2.5 Stockham

L'Algorithme de *Stockham* a pour but initial de supprimer l'étape de *bit reverse* en la répartissant progressivement par entrelacement avec les calculs.

Il a été référencé dans l'article de Cochrane et Cooley [12] en 1967 oú les auteurs attribuaient cet algorithme à T.G Stockham.

C'est un algorithme dit *auto-sort* qui permet de calculer la FFT à partir d'une séquence d'échantillons reçus dans l'ordre naturel, pour restituer le résultat dans le

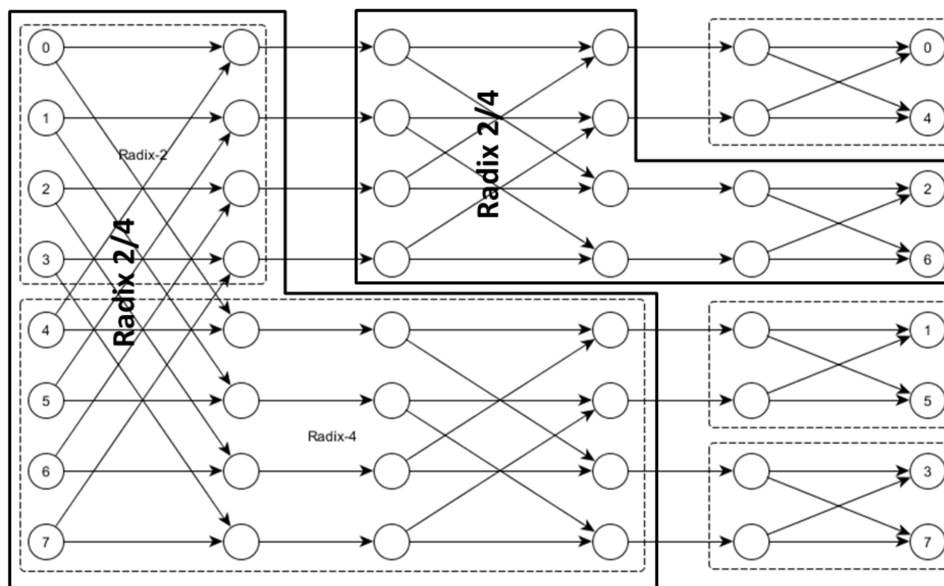


FIGURE 2.13 – *Signal-flow graph* split-radix FFT de taille 8

même ordre (voir figure 2.14).

L'étape de bit reverse est ainsi entrelacée avec les calculs.

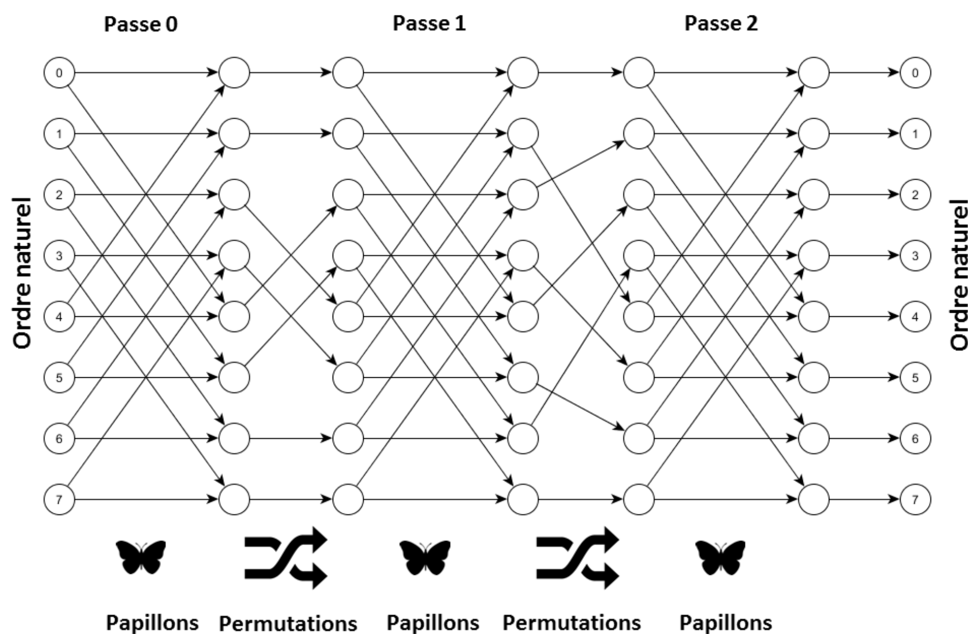


FIGURE 2.14 – *Signal flow graph* FFT selon l'algorithme de Stockham

La complexité de calcul d'un algorithme de *Stockham* reste strictement identique à celle d'un algorithme de type radix standard car le cœur de calcul de l'algorithme s'appuie sur le radix-2 ou 4. C'est juste l'organisation entre calculs et communications / transferts de données qui est modifiée, ce qui peut être potentiellement très intéressant pour la parallélisation de ces opérations.

L'algorithme de Stockham ne peut pas se faire en place (*in-place*), l'utilisation d'un tableau de données supplémentaire est nécessaire. Ce tableau « tampon » permet de changer la localité des résultats après chaque étage de calcul de la FFT. Le tableau initial et le tableau « tampon » alternent leur rôle après chaque étape. Ceci a pour conséquence de ne pouvoir garantir de retourner les résultats de la FFT sur le tableau initial pour toutes les tailles d'échantillons. Lorsque $log_2(N)$ n'est pas pair, il est nécessaire d'avoir recours à une copie de tableaux pour restituer les résultats de la FFT.

2.2.6 Autres versions

Il existe d'autres méthodes publiées dans la littérature, que nous ne considérerons pas dans notre travail, et mentionnées ici par soucis de complétude.

L'algorithme *Prime factor FFT* a été introduit par Good et Thomas en 1958 et 1963 [17] [18], il ré-exprime la DFT de taille $N = N_1 N_2$ en une DFT en deux dimensions $N_1 \times N_2$, pour ce faire N_1 et N_2 doivent être premiers entre eux. L'utilisation de cet algorithme n'est pas compatible avec une taille N qui est une puissance de deux.

L'algorithme FFT de Winograd [19] est considéré comme une dérivation du PFA(*Prime Factor Algorithm*). C'est un algorithme efficace connu pour sa faible complexité en multiplications [20].

2.2.7 Analyse et comparaison entre les versions

La comparaison entre les différentes versions de FFT a donné lieu à de nombreux travaux, tant théoriques qu'expérimentaux. Le site FFTW (`http://www.fftw.org`) recense actuellement la plupart de ces contributions.

Néanmoins ces travaux sont en général sensibles à la nature de l'architecture visée, ce qui justifie les travaux de cette thèse.

On considère en général en informatique la performance d'un processeur par le GigaFlops, défini comme le nombre d'opérations flottantes effectuées en une nanoseconde.

Dans la communauté de l'analyse de la FFT [1] on étend en général cette définition, et on parle de GFlops FFT en multipliant cette valeur par $5NLog_2(N)$, ce qui revient à normer les chiffres sur un algorithme abstrait (proche de la version de base Cooley-Tukey) qui comporterait exactement ce nombre $5NLog_2(N)$ d'opérations. Nous adopterons cette mesure dans nos résultats du chapitre 4.

2.3 Applications basées sur la FFT : exemple de détection Radar

La plupart des algorithmes de traitement du signal reposent sur une analyse spectrale du signal, et dans la majorité des cas leur implémentation utilise la transformée de Fourier rapide (FFT). Ainsi, optimiser la FFT, c'est aussi optimiser toute l'application (figure 2.15).

Une application populaire qui utilise massivement la FFT est le Radar à synthèse d'ouverture plus connu sous son acronyme anglais SAR (*Synthetic aperture Radar*),

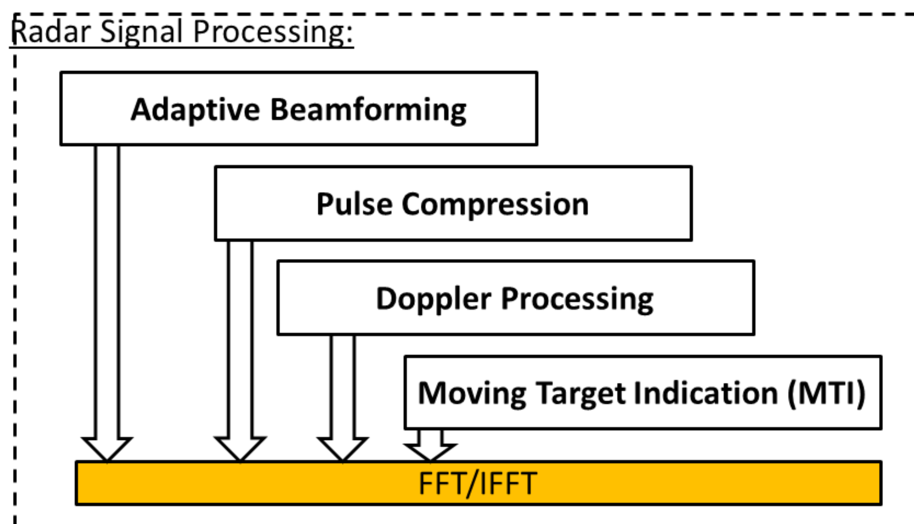


FIGURE 2.15 – Algorithmes traitement Radar qui utilisent la FFT comme bloc de base

ce type de radar est un radar imageur qui a pour but la télédétection principalement aérienne (dans le cadre de cette thèse).

Généralement les traitements SAR temps réel sont faits en utilisant des DSP (*Digital Signal Processor*) et/ou des FPGA (*Field Programmable Gate Array*). Du fait de la difficulté de programmation et du déverminage des DSP et FPGA, d'autres solutions ont émergé, à savoir celles basées sur les CPU ou les GPU [21].

Kontron privilégie dans le cadre de cette thèse la modularité et la portabilité de la solution SAR. La puissance de calcul fournie par les derniers processeurs d'Intel, à la fois au niveau CPU et GPU (intégré à la puce), rend cette solution applicable aux cas d'applications considérées par Kontron.

Plusieurs algorithmes de traitement SAR existent dans la littérature : *Extended Exact Transform Function* (EETF), *Chirp Scaling, Frequency Scaling, Omega-K, SPECAN* et *Range-Doppler Algorithm* (RDA). Ce dernier est l'implémentation la plus utilisée.

Les principales étapes du traitement SAR via l'algorithme *Range-Doppler* (RDA) consistent en une succession de FFT sauf pour l'étape du RCMC (*Range cell migration correction*) qui nécessite d'appliquer une interpolation sur le signal afin de corriger la forme hyperbolique de celui-ci, cette étape peut aussi être faite à l'aide de FFT avec la méthode introduite récemment par [22]. Cet algorithme est celui retenu pour notre application radar.

Le traitement SAR (RDA figure 2.16) nécessite aussi une étape communément appelée *corner turn*. Cette opération consiste à tourner les données bi-dimensionnelles (*Range et azimut*) à $90°$. Mathématiquement parlant, cette opération correspond à la transposition de la matrice des données (2.31).

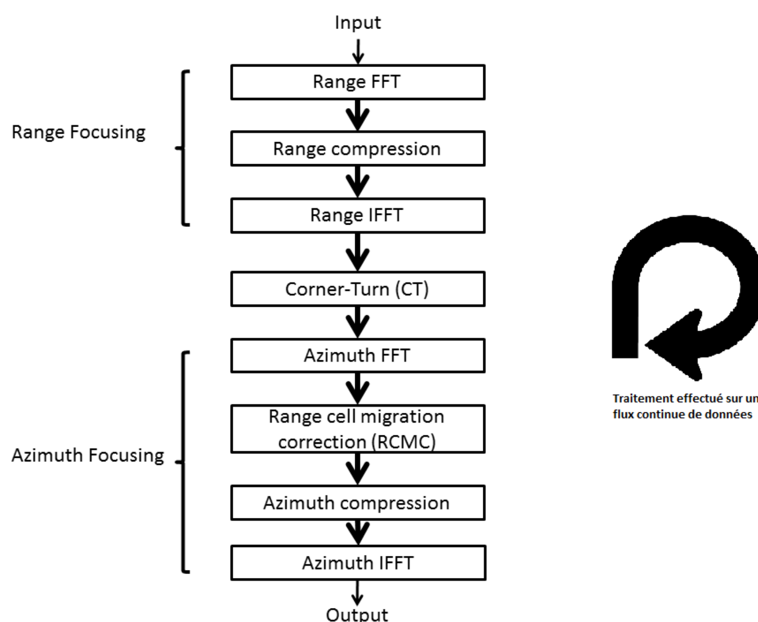


FIGURE 2.16 – Schéma bloc de l'algorithme *Range-Roppler* (RDA) pour le traitement SAR

$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{pmatrix} \quad (2.31)$$

Les données traitées par notre application sont sous forme de matrices de nombres complexes simple précision de taille 1024×1024. L'algorithme RDA est appliqué sur un flux continu de matrices de données 1024×1024. Nous appliquons le même traitement sur des données différentes à chaque itération de l'algorithme. Nous détaillerons cet algorithme dans le chapitre 5 de cette thèse.

Le traitement SAR nécessite une infrastructure de calcul intensif, les GPU offrent un très bon compromis par rapport aux autres alternatives. Dans notre domaine, à savoir celui de l'informatique embarquée, les critères taille, poids et énergie regroupés communément dans l'acronyme anglais SWaP pour (Size Weight and Power) sont des critères très importants. Un exemple simple est celui du radar SAR embarqué dans un drone : le poids influe sur l'altitude du drone, il influe aussi sur l'autonomie, donc dans un encombrement minimum il faut embarquer une puissance de calcul suffisante pour que le drone puisse voler assez vite et longtemps pour exécuter sa mission en général critique.

Trouver un algorithme optimal de la FFT pour une architecture cible n'est pas une chose triviale. L'ordonnancement des calculs et des mouvements de données afin de réduire le surcoût engendré par les mouvements de données (*register spill*) est un problème NP-complet[23]. Avec les architectures multi-cœurs et les niveaux supplémentaires de la hiérarchie mémoire, le problème est rendu encore plus complexe.

Bilan et discussion

Toutes les versions connues du calcul de la FFT s'organisent autour du bloc de base d'un papillon à deux entrées et deux sorties, dont le calcul doit s'effectuer de manière quasi-atomique avec des instructions dédiées (FMA). Nous avons proposé une amélioration incrémentale du calcul de ce bloc, qui augmente sensiblement en pratique la qualité des calculs flottants simple précision.

Pour récapituler, le calcul de la FFT procède en étages. Les $(n-1)$ premiers étages de l'algorithme calculent sur deux moitiés distinctes des valeurs une FFT de taille $(n-1)$, puis le dernier étage mélange ces résultats à nouveau par moitié, mélangeant dans un certain ordre la moitié des valeurs de la première « demi-finale » avec celles de l'autre.

Bien que l'on puisse envisager de nombreux ordres pour sélectionner les moitiés, on procède en général suivant l'ordre « naturel » qui combine d'abord les valeurs voisines sélectionnées deux par deux, puis élargit les distances en les doublant à chaque fois (voir figure 2.10), ou parfois suivant l'ordre inverse (combinant d'abord les valeurs les plus distantes, puis les rapprochant (voir figure 2.11).

On peut obtenir un code uniforme pour chaque étage en pratiquant effectivement les permutations qui « réalignent » les données dans une position semblable à chaque fois. On peut aussi *in-liner* le code de tous les étages à la suite, pour y pratiquer directement les changements d'indice (avec un code distinct alors pour chaque étage). La première solution ajoute le coût des permutations, mais la seconde fait gonfler la taille du code qui peut ne plus tenir en mémoire cache. Nous représenterons généralement ces permutations par une représentation graphique, alliant le graphe de dépendance (déplacement) des données entre deux étages, avec une signification donnée à l'ordre (vertical) des index des valeurs en mémoire.

En fait, nous allons être amenés dans nos travaux à considérer une approche mélangeant permutations explicites et *in-lining* du fait que nous allons trouver des limites au nombre d'opérations que l'on peut effectuer en n'utilisant que la mémoire très locale (registres) des processeurs SIMD sur CPU, ou des *Execution Units* sur GPU. Ces considérations de taille seront détaillées dans le chapitre suivant (pour le CPU et le GPU de notre architecture *Intel Core*), et la nature de nos solutions et ajustements des algorithmes (entre mouvements physiques de données et *in-lining* de code agrégeant des étages) feront l'objet principalement du chapitre 4.

On peut évidemment rappeler que le calcul de n étages de FFT (ou, de manière équivalente, d'une FFT de taille 2^n) nécessite n valeurs complexes, dont $2 \cdot n$ nombres flottants dans notre approche. C'est la taille qu'il nous faut chercher à ajuster à l'espace des registres.

Chapitre 3

Les architectures de calcul intensif

Motivations

Nous décrivons dans ce chapitre les tendances du parallélisme architectural existant au niveau élémentaire d'une puce matérielle unique (possiblement multicœur, voire multiprocesseur), différent du niveau macroscopique des grilles de machines et autres data centers.

Nous débutons par des généralités et rappels, puis nous nous concentrons sur le type de combinaison CPU-GPU que l'on trouve dans les architectures *Intel Core* (*Ivy Bridge* et *Haswell* notamment) qui formaient la cible de nos travaux au sein de *Kontron*. Le problème central consiste à bien comprendre et caractériser les tailles et les débits des différents composants de mémorisation et de transfert de données, ainsi que leur articulation, pour exploiter ces informations dans le placement optimisé des fonctions du calcul de FFT. Nous décrivons également certains aspects du langage OpenCL utilisé pour la représentation de nos algorithmes, et des mécanismes particuliers de compilation de ce langage sur ces architectures.

Dans le chapitre suivant nous étudierons le placement optimisé de la FFT sur cette architecture *Intel Core*. Il sera important de tester l'efficacité en performance, mais également en consommation et surtout en température (le circuit peut se mettre en mode dégradé, voire hors tension, s'il devient trop chaud). Nous abordons ces aspects en caractérisant le circuit de manière expérimentale, à l'aide des équipements disponibles au sein de la société Kontron.

3.1 Parallélisme "on-chip" (généralités)

3.1.1 Parallélisme d'instructions

Historiquement, dans les CPUs, chaque instruction est exécutée lorsque l'instruction précédente est terminée. Rapidement, les concepteurs ont vu qu'il était possible d'optimiser le flux d'instructions en divisant en plusieurs étages le traitement d'une instruction, ce qui améliore considérablement les performances du processeur.

Ces étages sont, par exemple dans le cas d'une machine RISC, lecture de l'opération en mémoire, décodage de l'opération en micro-opérations, lecture des opérandes, exécution et écriture du résultat.

Cette technique, nommée *pipeline*, permet de produire le résultat d'une opéra-

tion sur la durée de traitement d'un étage, celui de l'exécution, et non plus sur la durée de traitement de l'instruction. Ainsi, en régime constant, la vitesse d'exécution est considérablement augmentée. De plus, le compteur de programme peut commencer à traiter l'instruction à exécuter avant même que ses opérandes soient disponibles. C'est ici qu'apparaît la première forme de parallélisme qui exploite donc le parallélisme entre les différents étages.

Le *pipelining* est une des premières optimisations qui a permis d'exploiter le parallélisme d'instructions, d'autres techniques sont venues ensuite pour encore améliorer ce type de parallélisme à savoir :

— l'apparition des processeurs dits superscalaires qui sont capables d'exécuter plusieurs instructions simultanément parmi une suite d'instructions, ces derniers comportent plusieurs unités de calcul, et sont capable de détecter l'absence de dépendances entre instructions.
— l'exécution dans le désordre qui permet au processeur d'exécuter les instructions non pas dans l'ordre des instructions du programme mais de manière à optimiser l'utilisation des ressources matérielles, sans toutefois violer la dépendance des données.
— le renommage de registre vient comme une solution pour le problème de concurrence sur un même registre qui survient lors de l'exécution dans le désordre.
— la prédiction de branchement permet de prédire le résultat d'un branchement et rend ainsi plus efficace l'utilisation du pipeline, une autre solution est l'exécution spéculative qui permet d'exécuter en même temps différents branchement et après la fin de l'exécution on garde le résultat du bon branchement, au cout d'une duplication de la logique voire du pipeline.
— les accès mémoire et les entrées sorties dans le désordre.
— le *Prefetch* ou pré-lecture des instructions et des données.

3.1.1.1 FMA *fused multiply-add*

Une autre amélioration architecturale, présentée dans le chapitre précédent 2.1.3, appelé FMA ou *fused multiply-add*. Elle a été introduite par IBM en 1990 dans leur processeur POWER1.

Cette instruction permet d'exécuter une addition et une multiplication avec un débit équivalent à la fréquence d'horloge.

Elle permet aussi d'avoir une meilleure précision de calcul flottant par rapport à l'utilisation de deux instructions indépendantes, ceci est du au fait que l'arrondi n'est fait qu'une seule fois, c'est-à-dire à la fin de l'opération $a = a + b \times c$.

Ainsi, l'instruction FMA permet de doubler les performances du CPU. La FMA est très répandue dans les processeurs de nouvelle génération et dans tous les GPU.

3.1.1.2 SMT *simultaneous multithreading*

Cette présentation du parallélisme d'instruction ne sera complète qu'avec la description du *simultaneous multithreading (SMT)* ou d'après la nomenclature d'Intel l'*hyperthreading*[24].

Il s'agit d'une forme de *multithreading* qui permet le partage d'un cœur de processeur superscalaire entre plusieurs threads.

Les processeurs non SMT (figure 3.1), dotés de 4 unités d'exécution différentes, passent alternativement d'un *thread* à l'autre pour l'exécution des instructions. En revanche, les processeurs SMT (figure 3.2) peuvent allouer des unités d'exécution à des *threads* différents simultanément (schématisés ici en vert pour le *thread* 1 et en bleu pour le *thread* 2).

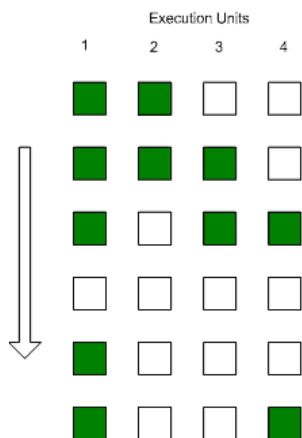


FIGURE 3.1 – Flux d'instruction classique

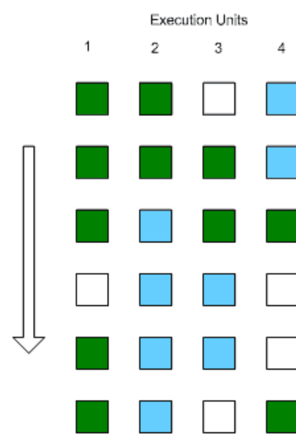


FIGURE 3.2 – Avec hyperthreading

Les registres utilisateur sont alors dupliqués et partagent les mêmes unités de calcul. Tout blocage d'un *thread* provoque un changement de contexte instantané (par exemple pendant un accès mémoire).

Le but du SMT est d'améliorer l'utilisation des ressources et ainsi exploiter au maximum le parallélisme d'instruction du processeur.

Le *multithreading* fournit en général pour un processeur, avec deux *threads* sur un cœur physique, 1.3 fois les performances d'un seul *thread*. Une autre technologie baptisée *"fused core"*[1]proposée par *Freescale* permet d'atteindre une performance entre 1.7 et 1.9. Ce gain de performance est dû à l'utilisation d'unités dédiées aux *threads* logiques.

Malgré les qualités de l'hyperthreading sur les applications grand public, nous déconseillons son utilisation pour les applications temps réel. Lors de nos essais, nous avons expérimenté sur l'architecture *Haswell* d'Intel à quatre cœur les performances d'une FFT 1D de 1024 points complexes avec l'*hyperthreading* inactif puis avec l'*hyperthreading* actif. Nous avons ainsi constaté une nette dégradation des performances comme le montre la figure 3.3.

La cause principale de cette dégradation est la pollution du cache L1 des cœurs par les threads concurrents sur les mêmes ressources. Il faut souligner aussi que dans notre cas l'essentiel des calculs tiennent dans le cache L1 qui est de taille 64KB.

Nous notons aussi que pour du code optimisé exploitant efficacement les unités d'exécution, par des dimensionnements convenables, l'*hyperthreading* n'apporte en général pas de gain de performances, il améliore surtout les situations où les calculs sont « creux » (par analogie avec les matrices creuses en mémoire).

Nous avons décidé que pour nos applications temps réel, pour des soucis de performances et de prédictibilité l'*hyperthreading* doit être désactivé. On verra au chapitre 4 que les différents cœurs d'un même processeurs seront par ailleurs plus tard

1. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=T4240`

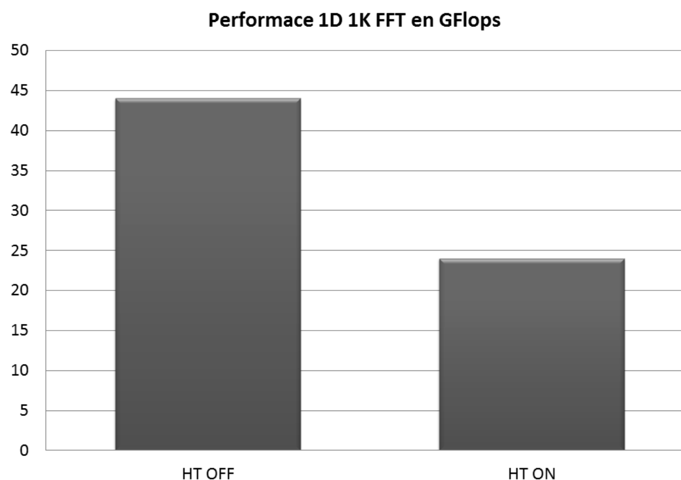


FIGURE 3.3 – Impact de l'hyperthreading sur la FFT : HT OFF les 4 cœur du CPU Haswell calculent un tableau de FFT de 1024 points complexes, HT ON les 8 cœur calculent le même tableau de FFT

disponibles alors pour exécuter chacune des FFTs distinctes de notre tableau de FFT de l'application Radar.

3.1.2 Le multicœur

Les processeurs actuels sont devenus des systèmes complexes capables d'effectuer plusieurs calculs simultanément, d'opérer sur plusieurs données à la fois, de prédire les différents branchements du programme, de changer sa fréquence dynamiquement ou encore de décider de l'ordre d'exécution des instructions. Toutes ces innovations ont été créées dans un seul et unique but, celui d'augmenter considérablement les performances des processeurs, but difficile à atteindre par l'augmentation seule de la fréquence. En d'autres mots, la fréquence représente la rapidité du processeur tandis que ces diverses innovations caractérisent l'efficacité de l'architecture.

Le traitement multicœur contribue à augmenter les performances et la productivité dans des ordinateurs de plus petite taille capables d'exécuter simultanément plusieurs applications complexes et de réaliser davantage de tâches en moins de temps.

Dans sa forme la plus petite, les processeurs multicœur sont constitués de deux cœurs identiques et d'une mémoire partagée pour les communications intra-cœurs.

Les architectures multicœur (voir figure 3.4) sont généralement dotées de cœurs homogènes, chaque cœur peut avoir un ou plusieurs threads SMT comme présenté en 3.1.1.2.

On peut alors espérer une augmentation des performances générale de l'application sur un processeur monocœur plus rapide (en fréquence) sans rien changer à l'application. Cependant, si nous gardons la même fréquence d'horloge du processeur mais que nous doublons le nombre de cœurs, le gain de performance n'est pas direct.

La loi d'Amdahl, énoncée par *Gene Amdahl* en 1967 [25], décrit très bien la limite

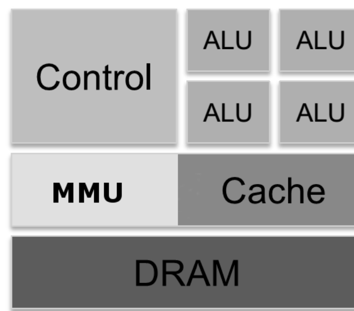


FIGURE 3.4 – Schéma simplifié d'un CPU à quatre cœurs

des performances que nous pouvons espérer sur une architecture multicœur.

Quand une application utilise du multithreading, le flux de contrôle de l'application peut être divisé en deux parties : les sections parallèles et les sections séquentielles. En pratique, l'application de cette loi se manifeste par une stagnation des performances au-delà d'un certain nombre de cœurs.

L'augmentation du nombre de cœur (plus ou moins nombreux et plus ou moins complexes) est désormais l'outil de prédilection des concepteurs de processeurs pour suivre le chemin du progrès.

L'ouvrage *Computer Architecture, A Quantitative Approach* de *Hennessy* et *Patterson*[26] dans sa 5ème édition, présente de manière détaillée ces architectures.

3.2 Modèle SIMD pour CPU

3.2.1 Principes et modèle de programmation

Le SIMD ou *Single Instruction Multiple Data* est une technique qui permet d'exploiter le parallélisme de donnée efficacement en amortissant le hardware de contrôle indépendant des données sur plusieurs unités de calcul (*processing elements*).

Plusieurs données sont regroupées et traitées de façon synchronisée (*lock-step*) en parallèle par une seule instruction.

Cette simplification hardware restreint ces éléments de traitement à avoir un flux de contrôle uniforme.

Un branchement où chaque élément de traitement branche vers une instruction différente provoque une divergence communément appelée *Branch divergence* dans le hardware SIMD. Ceci est généralement traité par une sérialisation de l'exécution ce qui résulte en une utilisation inefficace du hardware SIMD.

Une architecture SIMD consiste donc en un ensemble d'unités de calcul coordonnées par une unique unité de contrôle assurant un flux commun d'instructions. Chaque unité de calcul opère sur les données qui lui sont attribuées.

Ce type de parallélisme est dit parallélisme de données car la même instruction est exécutée sur des données différentes.

Nous pouvons voir sur la figure 3.5, que l'addition de deux tableaux de données A et B est effectuée à l'aide d'une unité SIMD de largeur 4. Elle exécute ainsi l'instruction d'addition simultanément sur quatre valeurs différentes des vecteur A et B, ensuite elle stocke le résultat dans le vecteur C.

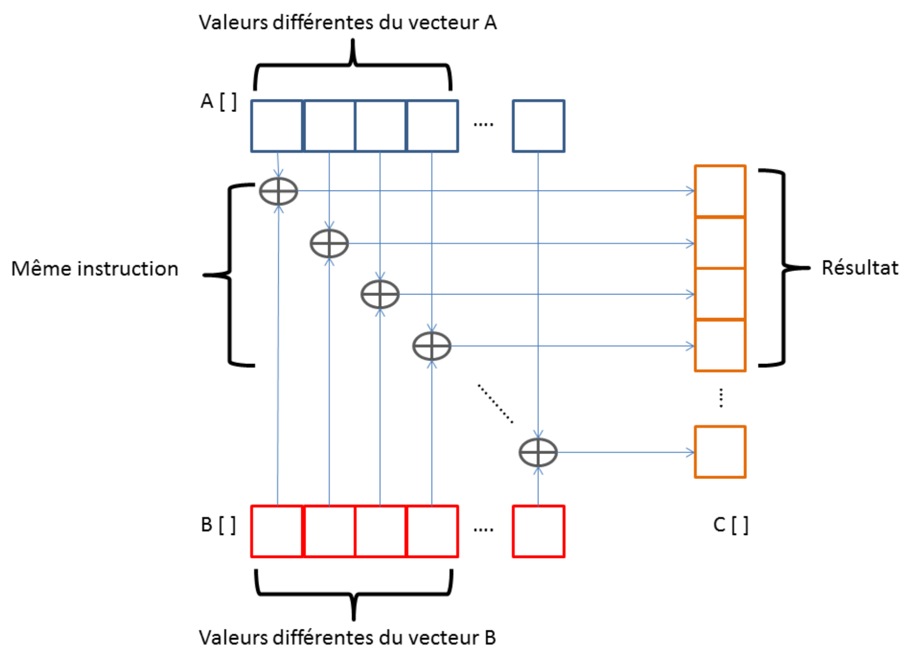


FIGURE 3.5 – Addition vectorielle de deux vecteurs $A + B$ en SIMD

Le bénéfice d'une telle architecture n'est plus à démontrer, notamment dans les applications graphiques où les calculs sur les données sont massivement parallèles. Elles sont aussi bien adaptées pour des implémentations d'algorithmes de traitement de signal où les formules mathématiques sont souvent exprimées sous forme de vecteurs et de matrices.

3.2.2 Mise en œuvre (intrinsics)

Les instructions SIMD sont notamment composées des jeux d'instructions suivants :

— Sur processeur x86 : MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2, AVX, AVX2 et AVX512.
— Sur processeur PowerPC : AltiVec.
— Sur processeur ARM : VFP, VFPv2, VFPv3lite, VFPv3, NEON, VFPv4.
— Sur processeur SPARC : VIS et VIS2.
— Sur processeur MIPS : MDMX et MIPS-3D.

Pour cette thèse, nous nous focaliseront sur le jeux d'instruction x86 d'Intel et plus précisément l'AVX2, du fait qu'il est le plus avancé (largeur 256bits) et le plus complet à l'heure où nous rédigeons ce manuscrit.

l'AVX ou *Advanced Vector Extensions* est une extension du jeux d'instructions x86 pour les microprocesseurs Intel et AMD, elle a été proposée par Intel en mars 2008 et supportée en premier par Intel dans le processeur Sandy Bridge en 2011 (figure 3.6).

l'AVX permet d'exécuter 8 opérations flottantes en même temps à l'aide de registres de 256bits.

Nous avons comparé les performances obtenues (figure 3.7) avec différentes générations d'instructions SIMD sur un programme FFT. Ce programme utilise la bi-

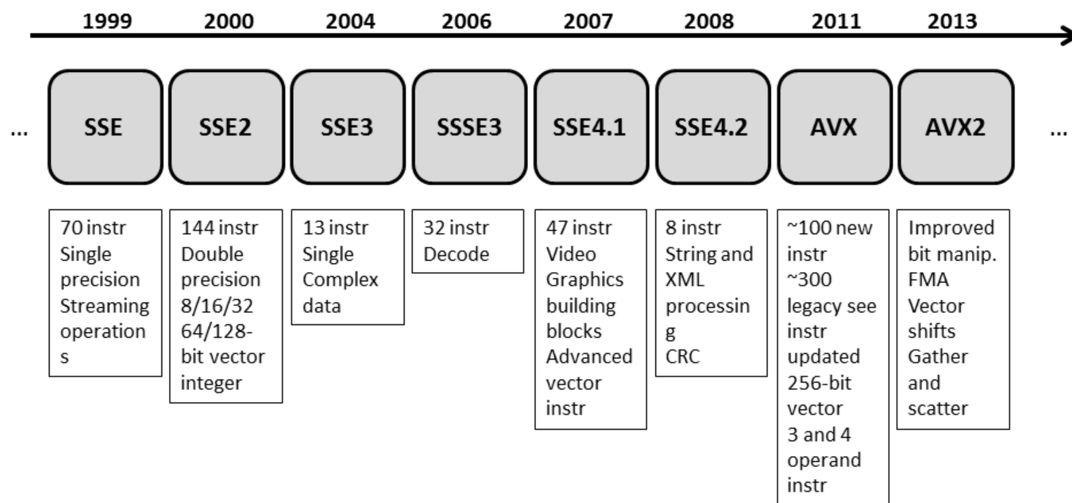


FIGURE 3.6 – Évolution des unités SIMD chez Intel

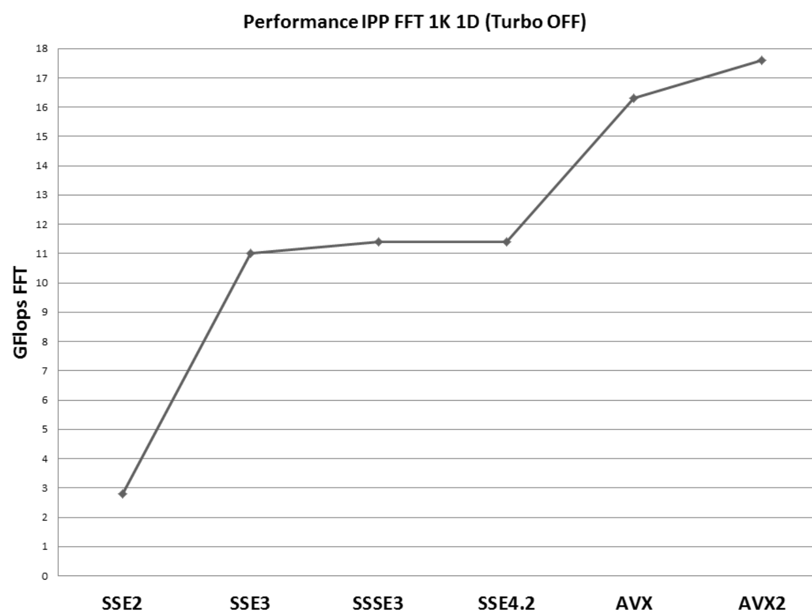


FIGURE 3.7 – Performances FFT obtenues sur CPU Intel Haswell en fonction du jeux d'instructions SIMD utilisé

bliothèque commerciale IPP d'Intel qui permet d'exécuter une panoplie d'algorithmes de traitement de signal dont la FFT.

Une remarque importante soulignée par la comparaison des performances est que l'augmentation de la largueur des unités SIMD a pour résultat un gain important des performances obtenues, par rapport aux autres améliorations d'architecture possibles (notamment l'évolution entre le SSE4.2 (128bits) et l'AVX (256bits)).

L'exploitation des unités SIMD se fait essentiellement par l'intermédiaire de trois méthodes :

— soit par compilation : c'est le compilateur qui se charge de vectoriser, néanmoins il n'existe pas encore de compilateur « magique » qui permette une vec-

torisation efficace des programmes.

— soit à l'aide d'*intrinsics* [2] : c'est un ensemble de fonctions en langage C qui représentent une correspondance 1 à 1 avec les instructions SIMD du processeur. L'utilisation des *intrinsics* permet d'écrire du code vectoriel depuis un langage de haut niveau (langage C par exemple). On doit noter aussi que dès qu'on utilise les *intrinsics*, le compilateur n'optimise plus ces portions de code. Le problème majeur de cette méthode est la non portabilité des performances, car si nous avons un code optimisé pour du SSE4.2 (largueur 128bits) et que nous voulons bénéficier des unités AVX, qui sont deux fois plus larges (256bits), nous devons réécrire le programme.

— soit par bibliothèque c'est-à-dire que l'application utilise des bibliothèques optimisées pour utiliser du SIMD comme le cas de la bibliothèque IPP.

La méthode efficace pour tirer profit de ces unités SIMD est l'utilisation des *intrinsics* C. Nous allons étudier dans cette thèse au chapitre 4 comment nous pouvons tirer partie de ces unités dans un cas d'étude concret, à savoir celui du traitement Radar.

3.2.3 Permutations de valeurs

On a vu en section 2.1.4 que des opérations rapides de permutation de données sont d'un intérêt critique pour l'efficacité de cette partie du calcul de la FFT. Nous détaillons donc les capacités de l'architecture SIMD *Intel Core* sur cet aspect, qui seront utilisées en section 4.1.

Les processeur *Intel Core* de troisième et quatrième génération sont dotés d'unités SIMD AVX, ces dernières utilisent un jeux de registres de 256bits chacun. Ces registres sont au nombre de 16 et sont nommés YMM (YMM0-YMM15).

Les instructions de permutation sont fournies par le jeux d'instruction. Comme cité précédemment, nous les utilisons à travers des *intrinsics* C. Pour effecteur la permutation de la figure 3.8 nous devons utiliser l'instruction assembleur em vpermilps à l'aide de l'intrinsic suivante :

```
1  __m256 _mm256_permute_ps (__m256 a, int imm8);  //vpermilps
```

L'entier imm8 sert de valeur de contrôle pour l'instruction, dans notre cas il contient la valeur 114 qui correspond à 01110010 en binaire.

Chaque instruction de permutation a une latence et un débit différent, il faut donc bien prendre en compte cet aspect pour l'adaptation de notre application.

Nous allons ici être confronté à un problème spécifique particulier : la largeur des instructions vectorielles SIMD du processeur considéré sera de 16, alors que, sans doute pour des raisons historiques de *legacy*, les permutations définies ne seront que sur des vecteurs de taille 8 (moitié moins). Il nous faudra donc combiner des opérations de cette taille, en optimisant leur durée, pour recomposer notre permutation de taille 16 désirée.

Nous mettrons en lumière en chapitre 4.1 une utilisation optimisée des instructions de permutation afin de produire une FFT hautement performante.

2. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

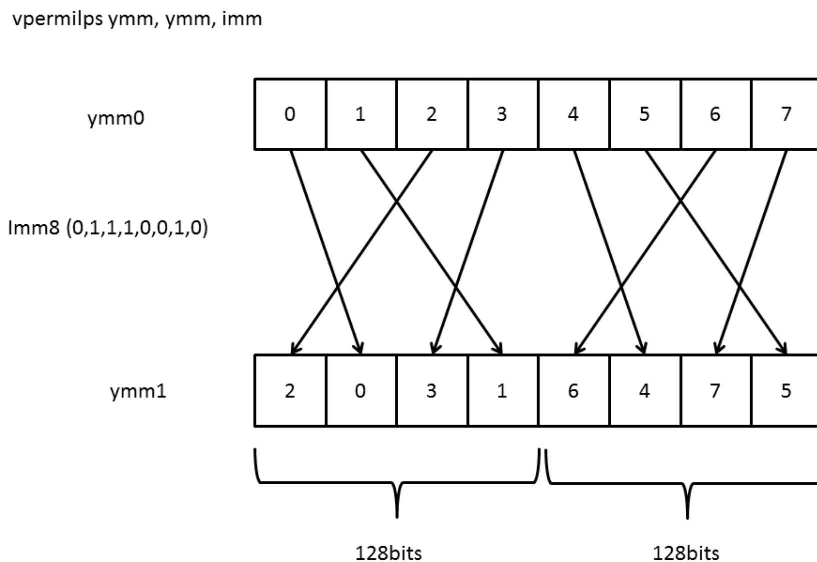


FIGURE 3.8 – Exemple de permutation SIMD en AVX

3.3 Modèle SIMT pour GPU

3.3.1 Évolution des GPU

Si les premiers GPU étaient à fonctions fixes, ils ont évolué pour devenir programmables. Ainsi, depuis la NVIDIA GeForce 3 qui implémente les *Pixels shaders* 1.1, les processeurs graphiques disposent d'une unité de géométrie programmable. Depuis la AMD Radeon R300 qui implémente les Pixels shaders 2.0, le calcul se fait sur des nombres flottants et plus seulement sur des nombres entiers.

On pouvait donc déjà créer des *Shaders*, des petits programmes permettant de manipuler des pixels ou des données géométriques (des Vertex). Pour traiter ces Shaders, la carte graphique incorporait des unités de traitement, capables d'exécuter des instructions sur des pixels ou des données géométriques.

Ces unités de traitement n'étaient ni plus ni moins que des processeurs assez rudimentaires, capables d'effectuer des instructions entières et flottantes. Ces dernieres n'étaient pas identiques : les instructions qu'ils étaient capables d'effectuer n'étaient pas les mêmes suivant que ces processeurs traitaient de la géométrie ou des pixels.

De nos jours, ces processeurs sont tous identiques (figure 3.9) et peuvent servir à faire aussi bien des calculs graphiques que des calculs sur des données quelconques.

Ces processeurs sont ce qu'on appelle des *streams processors*, des processeurs spécialement conçus pour exécuter des suites d'instructions sur un grand nombre de données. Sur ces processeurs, des programmes, nommés *kernels*, sont appliqués entièrement à un tableau de données que l'on appelle un *stream*. Dans les cartes graphiques actuelles, ce *stream* est découpé en morceaux qui seront chacun traités sur un *stream processor* ou *execution unit*. Chacun de ces morceaux est appelé *thread logique* ou *work-item*.

Les GPU modernes sont composés de *processing elements* (ou *stream processors* ou encore *execution units*) qui sont en réalité des sortes d'unités SIMD un peu plus sophistiquées, souvent appelées unités SIMT (*single instruction multiple data*), ca-

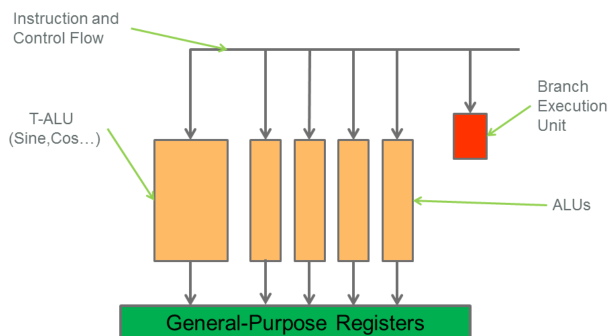


FIGURE 3.9 – Architecture d'un Execution unit GPU

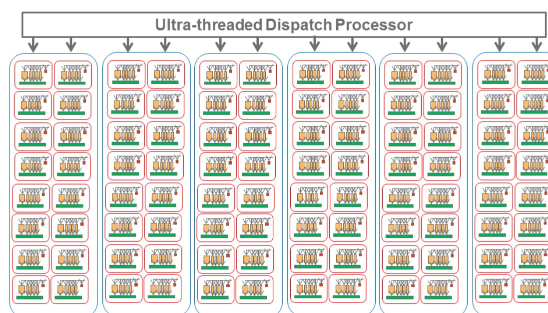


FIGURE 3.10 – Organisation des unités d'exécution du GPU d'AMD E6760

pables d'exécuter un certain nombre d'opérations flottantes en un coup d'horloge. Nous allons détailler dans la section suivante le principe de programmation SIMT.

Les GPU sont organisés de manière hiérarchique, ici en clusters de *processing elements* voir (figure 3.10). Ces clusters partagent entre eux un ou plusieurs niveaux mémoires / caches. Tous les GPU existants suivent la même structure, les noms peuvent changer mais l'organisation reste la même.

Ici encore, comme pour le cas des cœurs dans une architecture multicœur, nous choisirons d'implanter une FFT individuelle sur un EU unique, en se réservant de calculer en parallèle plusieurs FFT sur les différents EU du GPU. Les raisons de ce choix seront détaillées au chapitre 4.

3.3.2 Principes et modèle de programmation

Le SIMT pour *Single Instruction Multiple Data* est un terme introduit initialement par Nvidia [3] et repris ensuite par tous les autres constructeurs GPU.

Les GPU sont dotés d'unités d'exécution vectorielles SIMD (*Single instruction multiple data*). Ces unités sont exploitables à travers le modèle de programmation SIMT. Chaque voie "*lane*" de l'unité vectorielle SIMD correspond alors à un *thread logique* séparé. Les GPU possèdent une logique hardware SIMT [27] qui leur permet de détecter automatiquement si les différents *threads logiques* exécutent la même instruction.

Chaque voie possède ainsi son propre pointeur d'instruction (PC *program counter*) et les exécutent à l'aide d'un masque d'exécution.

Si tous les threads exécutent la même instruction (voir figure 3.11), alors on dit qu'ils sont cohérents.

Si un *thread* branche dans une direction différente où il exécute une autre instruction (voir figure 3.12), alors on dit qu'ils sont divergents.

Un seul séquenceur d'instructions suit l'état des threads divergents ou cohérents, il produit ainsi un masque qui active les éléments de traitement sur la même instruction pendant un cycle d'horloge.

Le SIMT est donc efficace quand les threads sont cohérents.

C'est le hardware avec son unité de contrôle qui détecte dynamiquement les différents threads demandant à exécuter la même instruction. Le SIMT peut aussi être

3. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

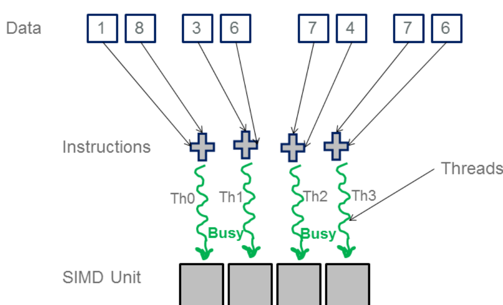


FIGURE 3.11 – Exécution SIMT en mode cohérent

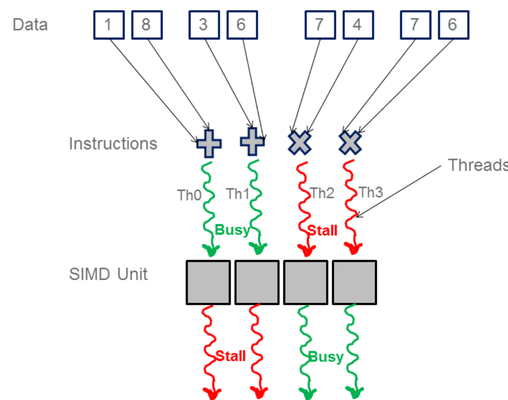


FIGURE 3.12 – Exécution SIMT en mode divergent

vu comme du *Vector lane Threading*.

Une des différences majeures entre les instructions vectorielle SIMD et le SIMT, c'est que le SIMD a besoin d'instructions intra-voie, comme les instructions de *shuffle* par exemple, ceci complexifie le jeu d'instruction SIMD et aussi l'implémentation hardware de ces instructions. Cependant, le SIMT n'a pas besoin de ces instructions car chaque voie appartient à un thread différent.

Une seconde différence est la gestion flexible des frontières de la largeur SIMD, du fait que l'utilisation de la largeur des vecteurs est dynamique en SIMT, ce qui n'est pas le cas en SIMD.

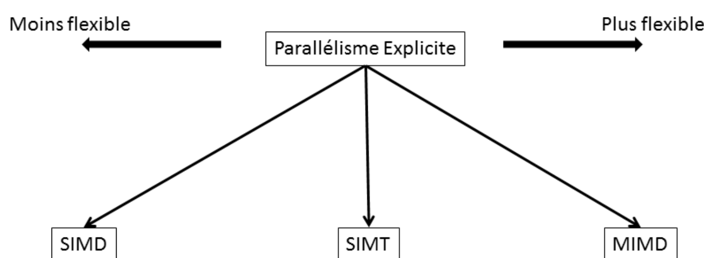


FIGURE 3.13 – Taxonomie en fonction de la flexibilité du modèle de programmation(SIMD, SIMT et MIMD)

Nous pouvons alors résumer les différences entre ces modèles de programmation à l'aide de la figure 3.13. Le SIMT est un modèle plus flexible que le SIMD en permettant une utilisation souple de la largeur de l'unité ALU vectorielle. En SIMT, les unités de calcul doivent être exploitées de façon synchronisées, ce qui n'est pas le cas pour le MIMD, sur du multicœur CPU par exemple, où deux threads s'exécutant sur deux cœurs CPU différents sont totalement indépendants.

3.3.3 Lien entre GPU et CPU, et transferts de données

Les GPU ne peuvent pas fonctionner en autonomie complète, ils doivent être alimentés par le CPU. Le CPU joue le rôle du maître et le GPU le rôle de l'esclave, ou en

d'autres termes, le GPU est un coprocesseur.

Les GPU possèdent leur propre espace mémoire, les données à traiter doivent y être transportées avant que le calcul puisse se faire. Ce transfert physique de données, souvent à travers une hiérarchie de mémoire, est la pénalité essentielle qui vient contrebalancer l'efficacité des calculs parallèles vectoriels étendus du coprocesseur GPU. À noter que les instructions (kernels et threads) doivent elles-aussi êtres chargées sur le GPU pour exécution : l'existence d'un cache d'instruction fait que de petits programmes assez répétitifs seront privilégiés sous cet aspect.

En général, les GPU sont catégorisés en deux grandes familles, à savoir GPU externe et GPU intégrés au CPU. Les GPU externes sont connectés via un Bus au CPU, généralement via le PCIexpress (avant en AGP) au CPU (voir figure 3.14).

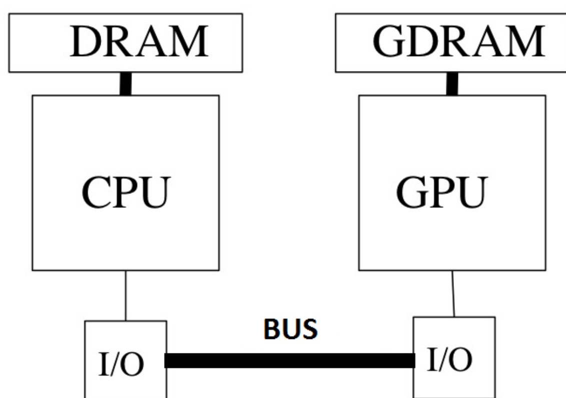


FIGURE 3.14 – Interconnexion classique CPU-GPU via un Bus

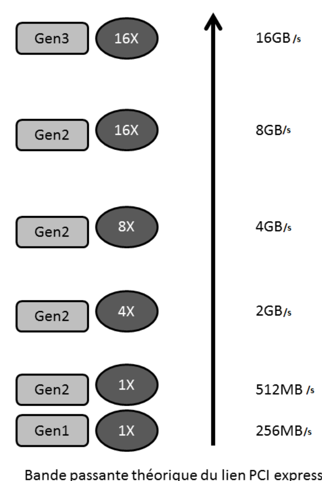


FIGURE 3.15 – Débits théoriques PCI express

Le PCI express fournit un débit de transfert assez réduit par rapport aux réseaux d'interconnexion internes au puces, en l'occurrence $16GB/s$ (théoriques) au meilleur des cas (voir figure 3.15).

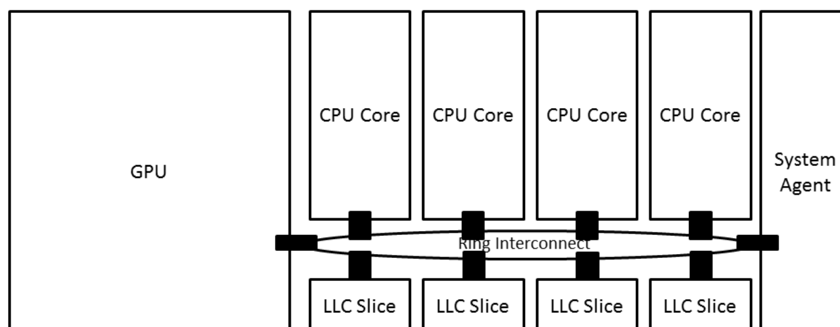


FIGURE 3.16 – Architecture GPU intégré

Les GPU intégrés au CPU (voir figure 3.16) partagent la mémoire DRAM, néanmoins la mémoire est partagée en zones dédiées. Le CPU doit copier les données à traiter par le GPU dans la zone mémoire affectée au GPU. Les transferts sont plus rapides que pour les GPU externes et bénéficient aussi du cache CPU.

3.3.4 Mise en œuvre (OpenCL)

Pour exploiter les GPU, nous utiliserons dans cette thèse le langage OpenCL.

OpenCL pour *Open Computing Language* est un standard ouvert développé par Khronos Group [4], il inclut à la fois un langage dérivé du C99 et une API pour programmer des systèmes parallèles hétérogènes, par exemple : CPU, GPU, DSP et FPGA.

OpenCL est le concurrent direct de CUDA de NVIDIA. Ce dernier est propriétaire et ne fonctionne que sur les processeurs graphiques NVIDIA.

Le modèle de programmation d'OpenCL repose sur une architecture maître et esclave où le CPU joue le rôle de maître et le GPU, par exemple, le rôle de l'esclave.

OpenCL permet d'exprimer à la fois du parallélisme de tâches et aussi du parallélisme de données. Il adopte le modèle SPMD (*Single Program Multiple Data*) pour exprimer le parallélisme, ainsi le compilateur se charge d'allouer et de lancer autant d'instances du programme (*kernel*) que de données à traiter.

Deux notions importantes dans OpenCL sont les **work-items** et les **work-groups**. OpenCL définit un **work-item** comme une instance logique du kernel, ou en d'autres termes, c'est un thread logique exécutant le code du kernel. Les work-items sont organisés en groupes afin d'être exécutés, on parle dans ce cas de **work-group**.

```
1 __kernel void vector_add(__global const int *A,
2                         __global const int *B,
3                         __global int *C)
4 {
5    // index de l'élément à calculer
6     int i = get_global_id(0);
7     C[i] = A[i] + B[i];
8 }
```

FIGURE 3.17 – Exemple *kernel* OpenCL qui additionne deux vecteurs d'entiers

Dans l'exemple suivant figure 3.17 le *kernel*, qui forme le programme qui s'exécute sur le GPU, additionne deux vecteurs d'entiers A et B et stocke le résultat dans le vecteur C. Pour pouvoir traiter un vecteur de 1000 entiers, le driver OpenCL (*runtime*) lance de manière implicite 1000 work-items qui exécutent ce *kernel*. Chaque work-item se voit attribué une valeur du vecteur à additionner. L'instruction OpenCL en ligne 6 permet à chaque work-item de connaître son numéro entre 0 et 999. C'est ce numéro qui est exploité pour calculer des données différentes par les work-items.

OpenCL définit aussi un modèle de hiérarchie mémoire, qui peut être décrit en trois niveaux :

— les registres : chaque cœur dispose de plusieurs registres, et chaque work-item accède à une partie, OpenCL nomme cette partie mémoire privée (*private memory*).

— la mémoire partagée : cette mémoire est partagée entre tous les cœurs, elle est aussi nommée mémoire locale (*local memory*).

— la mémoire globale : cette mémoire est accessible à tous les cœurs y compris les cœurs CPU, c'est la DRAM (*global memory*).

Une autre zone mémoire est aussi définie : la mémoire constante accessible en lec-

4. https://www.khronos.org/

ture seule et dédiée à stocker les données constantes. En effet, la plupart des matériels récents utilisent une mémoire cache spécialisée pour réduire la latence d'accès aux données constantes les plus utilisées.

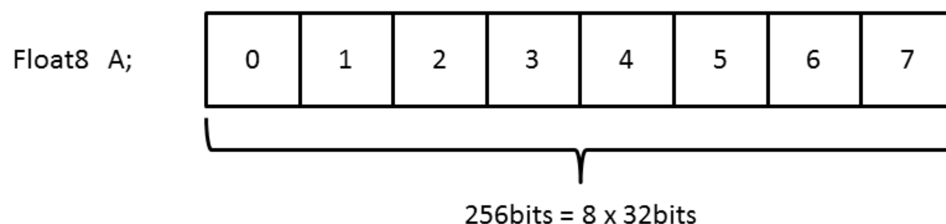


FIGURE 3.18 – Float8 est un vecteur de 8 flottants

Malgré la présence de types de données vectoriels dans OpenCL (par exemple float8 qui correspond à un vecteur de 8 flottants figure 3.18), ainsi que les opérations de manipulation de données vectorielles associées ; son utilisation reste liée à l'implémentation du compilateur faite par le constructeur du chip : par exemple chez *Intel*, l'utilisation d'un type vectoriel ne se traduit pas par une instruction vectorielle, comme nous pouvons le souhaiter, mais plutôt par une transformation scalaire du vecteur. L'utilisation des types vecteurs chez Intel sont plutôt adaptés aux accès mémoires, qui nécessitent un regroupement des accès mémoire (*memory coalescing*) afin d'avoir un meilleur débit.

CUDA	OpenCL	OpenCL Intel
Kernel	Kernel	Kernel
Thread	Work item	Channel
Warp	...	EU Thread
Thread block	Work group	Work group
Grid	Index space	...
Local memory	Private memory	registers(GRF)
Shared memory	Local memory	SLM L3
Global memory	Global memory	Global memory
Scalar core	Processing element	channel
Multiprocessor(SM)	Compute Unit	EU

TABLE 3.1 – Comparaison terminologie CUDA , OpenCL et OpenCL Intel

Dans la littérature OpenCL on trouve beaucoup de termes différents pour dire la même chose, en fonction du constructeur. Par soucis de clarté, nous allons utiliser le tableau 3.1 qui met en correspondance les terminologies CUDA et OpenCL.

Nous adopterons dans la suite de cette thèse la terminologie OpenCL d'Intel.

OpenCL est doté de mécanismes de synchronisation entre threads : barrières et

fonctions atomiques (*atomics*). Les barrières synchronisent les *work-item* d'un même *work-group*. Il n'est pas possible (en natif) de synchroniser les *work-groups* entre eux.

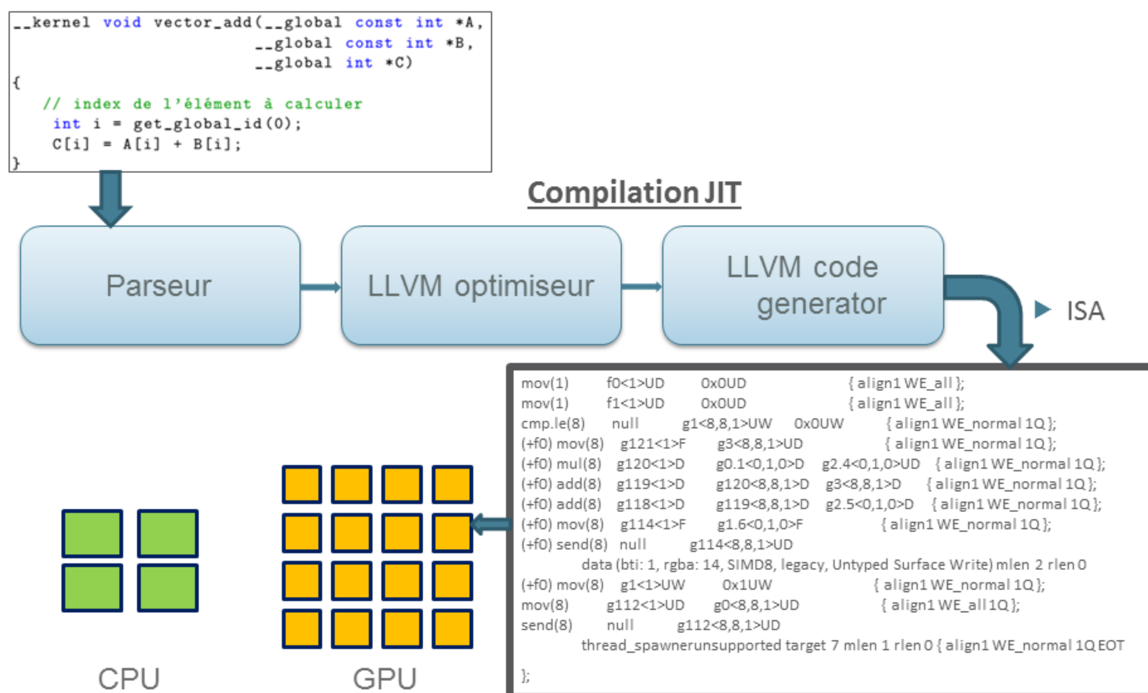


FIGURE 3.19 – Flot de compilation OpenCL sur GPU

Le code OpenCL est compilé par un compilateur JIT (*Just In Time compiler*) (figure 3.19) qui permet de traduire le source OpenCL en langage intermédiaire. Ce dernier est traduit en ISA (*Instruction Set Architecture*) GPU correspondant pour l'exécution sur la cible. Ceci permet d'avoir un même code source qui s'exécute sur différentes architectures GPU, sachant qu'il n'existe pas de compatibilité binaire entre les architectures GPU comme c'est le cas pour les CPU x86.

3.4 Architecture *Intel Core* considérée chez Kontron

3.4.1 GPU Intel

Les processeurs Intel actuels sont devenus des SoC (System-on-a-Chip) car ils intègrent sur la même puce deux, quatre ou huit cœurs CPU, ainsi qu'un processeur graphique GPU et même éventuellement d'autres unités spécialisées.

L'architecture des processeurs graphiques d'Intel est bâtie sur un principe de modularité, pour permettre de dériver plusieurs modèles des mêmes briques de base.

L'organisation des GPU Intel actuels ajoutent un niveau à la structure traditionnelle EU -> CU -> GPU, en les nommant (du plus petit au plus grand) : Execution unit

, *SubSlice*, *Slices*, GPU (voir figure 3.20).

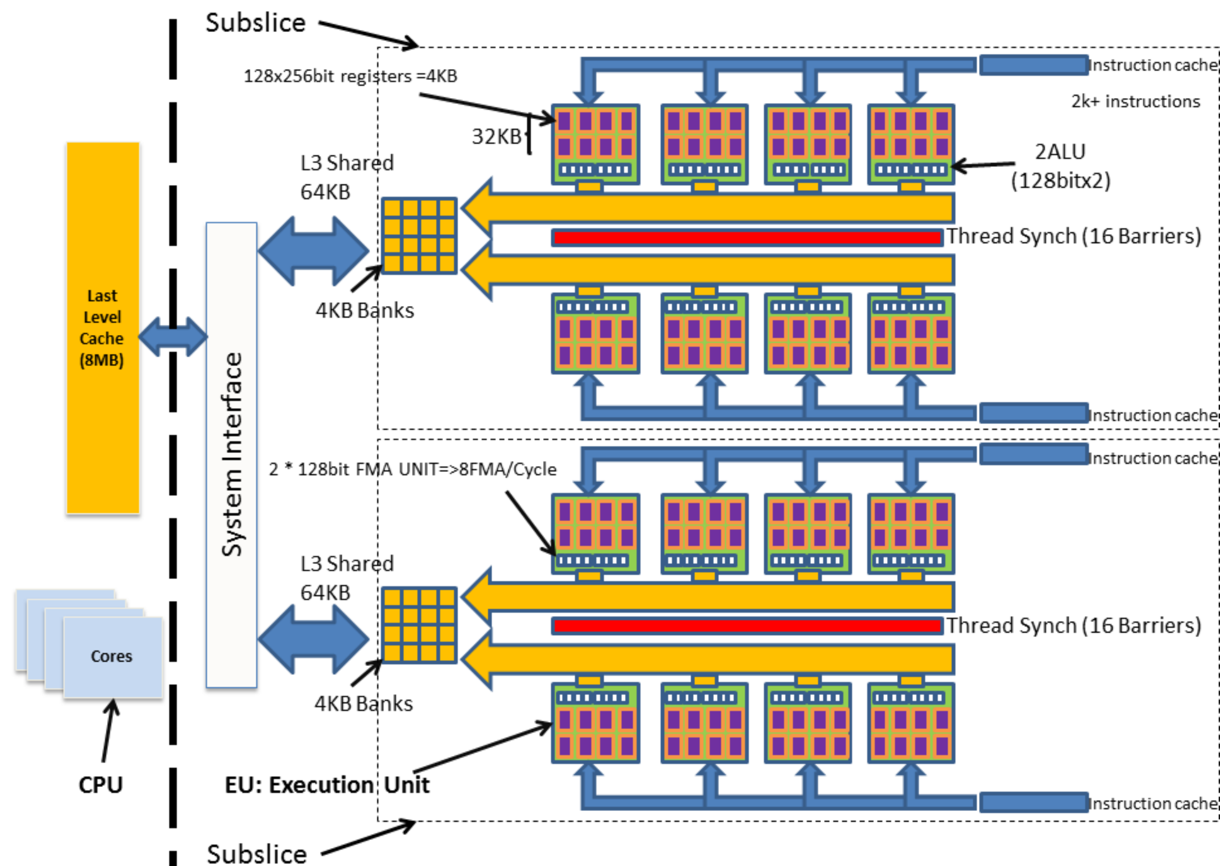


FIGURE 3.20 – Architecture GPU Intel IvyBridge HD4000

L'interface du processeur graphique avec le reste des composants du SoC est assurée à travers une unité d'interface qui porte le nom de GTI (*Graphics Technology Interface*). Cette interface implémente un système de gestion de l'énergie (*power management*) pour le processeur graphique et fait aussi le lien entre les domaines d'horloges (voir figure 3.22) qui sont d'habitude différents par rapport au reste du SoC.

Chaque *slice* possède une infrastructure logique dédiée aux barrières de groupes (work-group). Cette logique est disponible comme une alternative hardware aux barrières gérées par compilation.

L'architecture GPU Intel peut supporter simultanément 16 work-groups actifs par subslice.

Chaque *slice* est dotée d'une suite d'opérations atomiques pour la lecture, la modification et l'écriture, soit dans la mémoire cache L3 soit dans la mémoire locale partagée.

Le commande streamer (CS) permet d'envoyer le flux de commandes provenant du driver vers les unités correspondantes à ces commandes. Pour les commandes de calcul, l'unité *global thread dispatch* est responsable de la balance de charge entre les unités de calcule du processeur graphique. Cette unité fonctionne en concert avec les *thread dispatch* locales dans chaque *subslice*.

Le Global thread dispatch fonctionne en deux modes : pour les *kernels* qui n'uti-

lisent pas de barrière de synchronisation hardware ni de mémoire locale partagée, il dispatche la charge sur tous les slices, afin de maximiser le débit et l'occupation. Pour les *kernels* qui utilisent les barrières ou la mémoire locale partagée, il assigne les work-groups à des *subslices* spécifiques, ceci afin d'assurer une localisation des barrières et aussi de la mémoire locale partagée dédiée à chaque *subslice*.

3.4.2 Unité d'exécution

Un cœur GPU ou selon la nomenclature Intel *Execution Unit (EU)*, est un *Simultaneous Multi-Threading* (SMT) processeur. Il possède plusieurs unités arithmétiques et logiques (ALU) de type SIMD pipelinées, pour un haut débit de calcul flottant et entier. La nature hautement multithreadée des EU, assure un flot continu d'instructions ; afin de masquer les latences des opérations longues tel que les accès mémoires (*scatter*/*gather*), les communications ..., etc.

L'exploitation de ces unités d'exécutions se fait à travers le modèle *SIMT*.

Pour des raisons de clarté, nous adopterons dans cette thèse le vocabulaire suivant : nous parlerons de *thread hardware* pour désigner les threads SMT de l'EU. Nous utiliserons *thread software* ou *work-item* pour désigner un thread OpenCL qui fonctionne sur une partie de l'unité SIMT (voir figure 3.21).

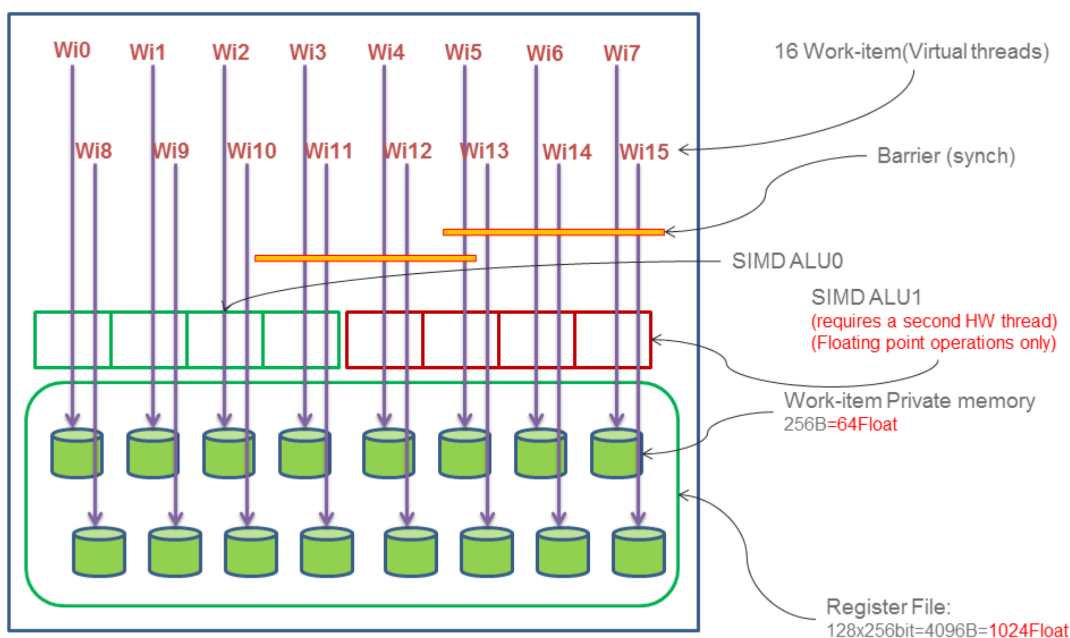


FIGURE 3.21 – Mapping des Work-items dans une EU GPU d'Intel IvyBridge

Chaque Thread hardware possède 128 registres de 256bits de large. Ces registres sont des registres généraux (GRF pour *general purpose register file*). Chaque EU a 8 *threads* hardware pour un total de 32KB de registres GRF. Des modes d'adressage

flexibles permettent d'adresser plusieurs registres, afin de former de plus gros registres, ou pour représenter des blocs rectangulaires de données. D'autres types de registres existent, ce sont des registres architecturaux (ARF *architecture register file*). Ils sont dédiés pour chaque thread hardware afin de sauvegarder l'état du thread, y compris sa pile de pointeur d'instruction.

Une EU peut exécuter à chaque cycle 4 différentes instructions qui viennent de 4 threads hardware différents. Un arbitrage est fait pour distribuer les instructions sur un des quatre slots d'exécution, cependant il est nécessaire que les quatre instructions proviennent de quatre threads hardware différents.

Les instructions de branchement sont dispatchées vers une unité de branchement dédiée. Les opérations mémoire ainsi que les instructions de communication sont dispatchée via une unité de *message passing* (Send Unit).

Dans chaque EU, deux FPU SIMD (*floating point units*) sont chargées d'exécuter les calculs flottants ou entiers. Ces unités sont capables d'exécuter quatre opérations flottantes 32bits en parallèle. Elles sont aussi capables d'exécuter à chaque cycle une multiplication addition (FMA). Donc chaque EU a une capacité d'exécution de 16 opérations flottante 32bit par cycle : $(add + mul) \times 2FPUs \times SIMD4 = 16$.

Une des FPUs supporte les fonctions mathématiques transcendantes (sin, cos,...,etc), ces fonctions sont certes plus rapides que leurs homologues logiciels, mais dans notre cas nous privilégions le pré-calcul des coefficients "*twiddle factors*" de notre algorithme FFT .

Les EU sont optimisées pour des données 32bits. Aussi le jeu d'instructions (*ISA : Instruction Set Architecture*) permet un support flexible en fonction de la largeur du SIMD. Effectivement, les unités SIMD peuvent être utilisées en largeur 1, 2, 4, 8, 16 ou carrément 32. Le choix est laissé à la charge du programmeur afin d'optimiser au mieux l'utilisation des registres en fonction du nombre de threads software (work-items) à lancer et aussi la quantité de données à traiter. Le compilateur Intel appelle ces modes de compilation SIMD8 , SIMD16 ou SIMD32, le premier mode SIMD8 correspond à un mapping 1 à 1 entre un *"lane"* SIMD et un *work-item*. Le mode SIMD16 correspond à l'exploitation de l'unité ALU SIMD en *double-pumped* c'est à dire nous pouvons mapper 16 *work-items* sur l'unité SIMD qui est de largeur 8 à la base. Ce deuxième mode est plus efficace que le premier car il permet d'avoir une meilleure occupation des unités, surtout lors d'accès mémoires longs. Le mode SIMD32 n'est pas pris en charge par toutes les générations de GPU Intel, il correspond à l'utilisation de l'unité d'exécution en *quad-pumped*. Le compilateur est ainsi en charge de générer le code SIMD afin de mapper efficacement plusieurs instances du noyau de calcul (*Kernel*) qui sera exécuté simultanément par un ou plusieurs threads hardware.

Nous notons aussi que les GPUs *Intel Core* définissent une taille maximale pour le nombre de *work-items* par *work-group*, cette taille est de 512 *work-items*.

En cas de branchement, l'unité de branchement est en charge de générer un masque d'exécution qui permet d'indiquer les instances du *kernel* qui doivent s'exécuter. Si toutes les instances du kernel exécutent la même instruction, alors les unités d'exécution SIMD sont utilisées efficacement.

3.4.3 Hiérarchie mémoire

Les GPUs sont caractérisés par une hiérarchie mémoire spécifique. Les tailles et débits mémoire sont très importants dans ce genre d'architectures, du fait des mouvements fréquents de données.

Chaque *subslice* possède son propre cache d'instruction et sa propre unité de dispatch de *threads*, ainsi que son propre *sampler* et un port de données provenant de la MMU. Nous notons aussi que le sampler est une unité de rapatriement de la mémoire en lecture seule, elle est plus utilisée pour les textures et les images. Le sampler est doté d'un cache L1 et L2 (voir figure 3.22), ainsi que de plusieurs fonctions fixes pour effectuer différents filtrages et transformations d'adresses pour les images.

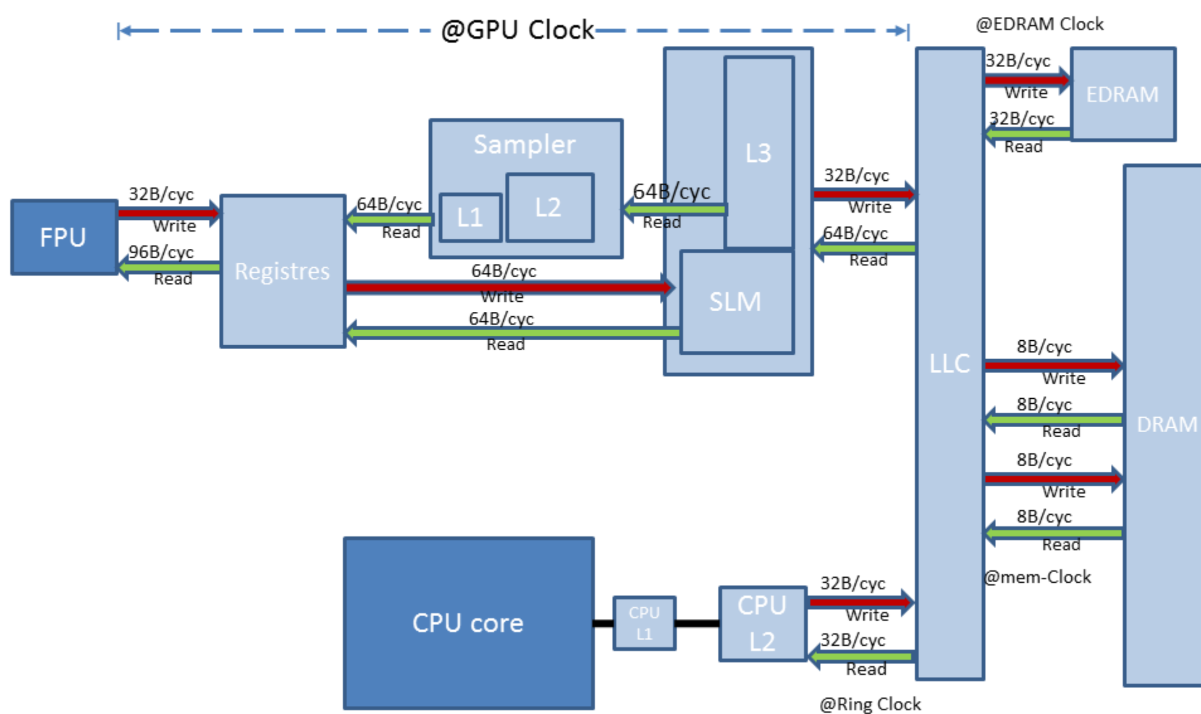


FIGURE 3.22 – Architecture mémoire du GPU Intel avec les débit de transferts

Le port de données est une unité de chargement et de stockage mémoire qui supporte les opérations de lecture et écriture efficace pour une variété d'accès génériques aux *buffers*, les opérations de scatter/gather SIMD, ainsi que les accès à la mémoire locale partagée. Afin de maximiser le débit mémoire, l'unité regroupe (*coalesce*) dynamiquement les opérations mémoire dispersées dans moins d'opérations, à travers une demande de ligne de cache non-dupliquée de taille 64Bytes.

Les instructions SIMD-16 sont très adaptées pour cette architecture, car les registres sont de taille 32Bytes, et le jeu d'instruction est flexible. Il permet d'utiliser deux registres adjacent comme si il s'agissait d'un seul registre plus grand de taille 64Bytes. Ce qui permet à une instruction SIMD-16 d'utiliser un opérande de taille

64Byte, en le faisant transiter par le cache L3 au travers du bus de données de largeur 64Bytes. Nous savons aussi que les lignes de cache L3 font 64Bytes de large, et le bus de données vers la LLC du SoC est aussi de largeur 64Bytes (voir figure 3.22).

Quelques modèles de processeurs Intel, notamment l'Intel *Haswell* possèdent une mémoire EDRAM (*embedded DRAM*) de 128 MB. Elle sert au processeur graphique comme un grand "*victim cache*" derrière le cache de dernier niveau (LLC).

Chaque sampler et chaque data port possède sa propre interface mémoire vers la L3, avec un débit lecture et écriture de 64Bytes par cycle par port de données.

Chaque donnée entrante ou sortante des samplers et des ports de données passent par le cache de données L3. Le débit de lecture et d'écriture dans le cache L3 est meilleur si les données sont alignées ou adjacentes dans la ligne de cache. Il faut savoir que les instructions passent aussi par le cache L3.

Un slice inclut une mémoire cache L3 organisée en bancs, ainsi qu'une petite mémoire locale partagée. La mémoire locale partagée est une sous partie du cache L3 dédié, elle est gérée par le programmeur. Elle permet le partage des données entre différentes instances du kernel. Cette mémoire est fortement divisée en bancs distincts, plus que le cache usuel L3. Elle permet des accès non alignés sans pénalité de performances. La taille de cette mémoire est de 64KBytes par *subslice*. Une restriction à mentionner pour l'utilisation de cette mémoire est que un *work-group* (OpenCL) ne peut partager les données que dans un *subslice* et dans une seule partition de mémoire partagée.

3.4.4 Interconnexion

La communication entre les cœurs CPU, les mémoires cache et le processeur graphique se fait par l'intermédiaire d'un anneau bidirectionnel de largeur 32Bytes pour les données, avec des liens séparés pour le contrôle (demande, acquittement, *snoop*). La MMU est aussi connectée à l'anneau.

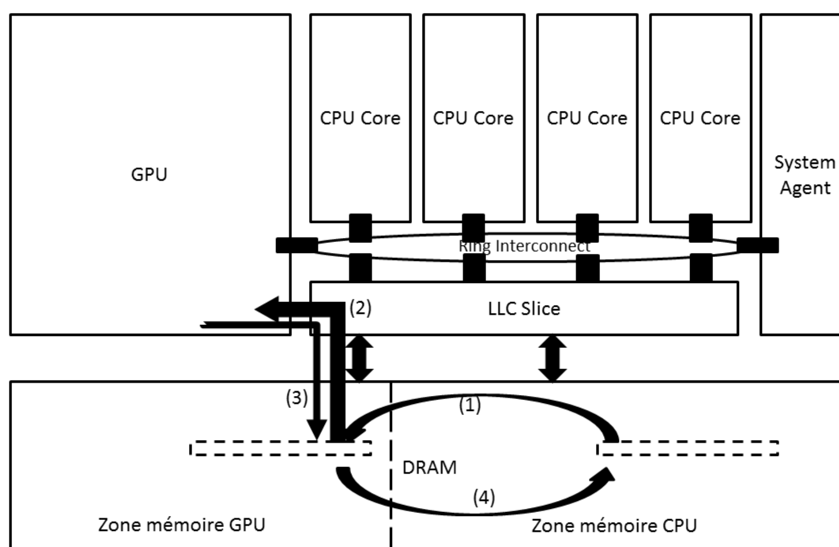


FIGURE 3.23 – Interconnexion CPU-GPU Intel SoC

Une des particularités des architectures GPU intégrées au CPU est la zone mé-

moire DRAM partageable entre CPU et GPU (voir figure 3.23) mais dédiée pour le CPU et pour le GPU, ce qui pousse à devoir faire une copie explicite des données dans la RAM entre la zone mémoire CPU et la zone mémoire GPU.

Les transferts mémoire de et vers la mémoire RAM bénéficient des différents niveaux de caches existant dans l'architecture (voir la section précédente 3.4.3). Néanmoins ces caches rendent la performance moins prévisibles, et une motivation pour nos travaux sera d'organiser les dimensionnements des calculs et des données afin d'assurer la localité dans les caches autant que possible.

Après chargement des données dans la mémoire du GPU (cf(2) figure 3.23), les unités d'exécutions EU se partagent la lecture et l'écriture des données résidant dans la mémoire locale partagée (SLM). Une synchronisation entre *work-items* est nécessaire pour garantir l'intégrité des données.

Nous avons vue dans la section précédente 3.4.3 que les mémoires SLM sont structurées en bancs, cependant du fait que les GPU sont des architectures massivement parallèles, nous devons éviter d'avoir des conflits de bancs mémoire. Ainsi, si l'on indexe nos accès mémoires d'une manière à maximiser le nombre de bancs utilisés, nous pouvons garantir une exploitation optimale du débit de transfert interne fourni par l'architecture. La section suivante 3.5 exercera ces transferts mémoire.

3.5 Mesures expérimentales sur les architectures *Intel Core*

Au sein de la société Kontron nous bénéficions d'environnements de développement (*design kits*) sur les processeurs *Intel Core* qui permettent d'analyser finement les comportements fonctionnels et extra-fonctionnels (power, température) des chipsets. Ces équipements (ainsi que des méthodes de mesure logicielles plus standard) nous ont permis de caractériser un certain nombre de phénomènes à prendre en compte dans la suite de cette thèse. Nous les décrivons dans cette section. Tous ne sont pas reliés entre eux, mais parfois des liens indirects apparaissent.

L'architecture GPU *Intel Core* est très complexe à prendre en main, du fait de la documentation ambiguë d'*Intel* et de l'outillage assez pauvre fourni par le constructeur. La phase de caractérisation "*micro-benchmarking*" est aussi nécessaire pour bien appréhender toutes les caractéristiques de cette architecture et bien comprendre les subtilités de son utilisation.

3.5.1 Débit de calcul maximum

L'une des premières questions qu'on peut se poser quand on veut étudier une architecture est : qu'elle est la performance de calcul maximum qu'on peut atteindre sur cette plateforme ? Est-il possible d'atteindre la performance maximum présentée sur la *Datasheet* du constructeur ? Est-elle juste une performance théorique ?. Cette question est importante pour savoir utiliser ultérieurement une valeur légitime pour la durée des calculs (avant même que les phénomènes de mouvements de données ne soient intégrés).

Pour répondre à cette question nous avons écrit un *micro-benchmark* en OpenCL (voir *listing* figure 3.24), qui permet de viser le maximum de performances (en GFlops) possible sur une architecture CPU ou GPU donnée (munie de l'instruction FMA).

Notre *Benchmark* repose sur une suite d'instructions de multiplications additions afin d'exercer au maximum les unités de calculs.

```
#define mulAdd()\
        x=mad(y,z,w);\
        w=mad(z,y,x);\
        x=mad(y,z,w);\
        w=mad(z,y,x);\
        x=mad(y,z,w);\
        w=mad(z,y,x);\
        x=mad(y,z,w);\
        w=mad(z,y,x);

__kernel void MAXFlops(__global float* res ,int inc)
{
        int tid=get_global_id(0);
        __private float x,y,z,w;
        x=0.0f;
        y=1.0f;
        z=1.0f;
        w=1.0f;
        for(int i=0;i<inc;i++)
        {
                //8 x 8 muladd = 64 mad= 128 op
                mulAdd();
                mulAdd();
                mulAdd();
                mulAdd();
                mulAdd();
                mulAdd();
                mulAdd();
                mulAdd();
        }
        //to avoid the compiler do clever things
        res[0]=w;
}
```

FIGURE 3.24 – Benchmark *Kernel* OpenCL GFlops maximum

Ce *benchmark* nous donne ainsi une indication sur les performances atteignables en pratique. Nous prenons cette référence pour définir l'efficacité de calcul dans cette thèse :

$$efficiency = \frac{perf_{mesuredPeak}}{perf_{theoriticalPeak}} \tag{3.1}$$

Par l'exécution de notre *micro-benchmark* et la mesure des résultats pratiques rapportés aux performances théoriques, nous observons en figure 3.25 une efficacité de calcul de l'architecture GPU *Intel HD4000* de 80%, et pour le GPU *AMD E6760* de 85%.

L'autre question intéressante est : comment ces performances crête passent-elles à l'échelle par rapport au nombre d'EU ? Sont-elles linéaires ou y'aurait-il des restrictions qui pourraient changer les résultats ?

Nous avons répondu à cette question en exerçant les unités de calculs de chaque EU du GPU *Intel IvyBridge HD4000* à l'aide du *benchmark* précédant en activant les unités d'exécution du GPU séparément. Nous avons ainsi pu constater (voir figure 3.26) que le passage à l'échelle des performances se fait sans problème (dans

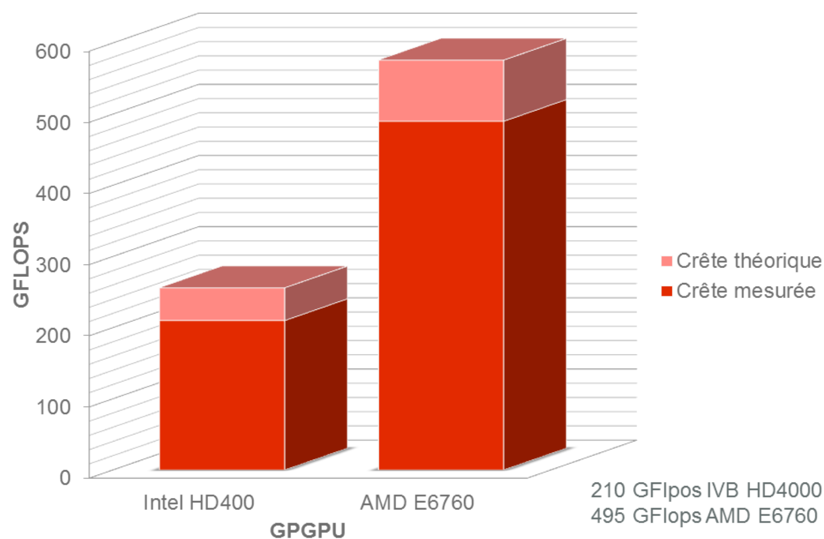


FIGURE 3.25 – Les performances crête mesurées par notre outils sur le GPU d'Intel HD4000 et le GPU AMD E6760

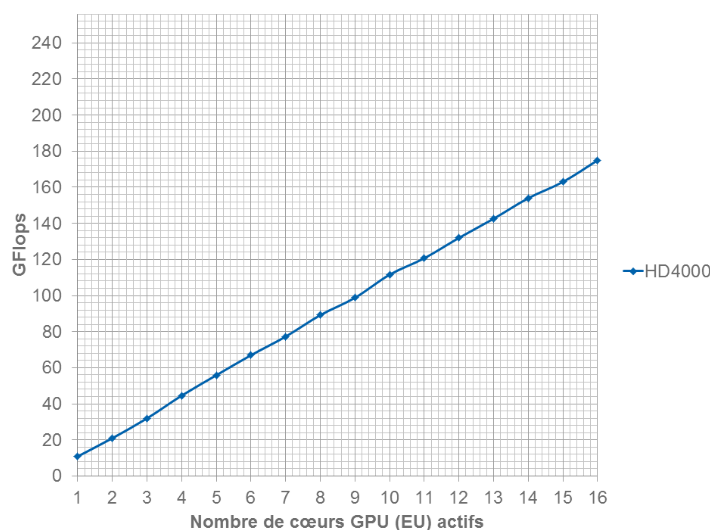


FIGURE 3.26 – Passage à l'échelle des performances sur le GPU Intel HD4000

une configuration où les cœurs EU travaillent indépendamment de manière autonome).

3.5.2 Taille des *work-groups*

Comme nous l'avons rappelé dans ce chapitre, le nombre de threads disponibles et utilisés est contrôlé en OpenCL par la déclaration de la taille du *work-group* (avec l'équation $work_group \times work_item = global_work_size$). Alors comment doit-on choisir la taille de nos *work-groups* ? Influencent-ils les performances ?

Nous avons choisi de mesurer l'impact de la taille du *work-group* sur les transferts

mémoires et sur les performances de calcul.

3.5.2.1 Débit de transfert mémoire et *work-groups*

Nous avons testé le débit de transfert mémoire obtenu depuis la mémoire RAM DDR3 jusqu'aux registres de chaque unité d'exécution EU du GPU *Intel HD4000*. Nous cherchons aussi à savoir à travers ce test comment ce débit se met à l'échelle en fonction du nombre de *work-items* utilisés. Nous faisons alors varier la taille des *work-groups* ainsi que le nombre total de *work-items*.

Pour ce faire nous avons utilisé comme *benchmark* le *kernel* OpenCL (voir figure 3.27), qui recopie une partie des données issues de la mémoire globale dans les registres GRF de chaque work-item (thread logique).

```
1  __kernel void ddrtoregisters(__global float* ddr,
2                               __global float* result,
3                               int N)
4  {
5          ulong i=0;
6          int off=get_global_id(0)*64;
7          ulong size=N;
8          //Maximum size for one thread is 64
9          __private float reg[64];
10         for(i=0;i<size;++i)
11                 reg[i]=ddr[off+i];
12         //to avoid the compiler from doing clever things
13         result[0]=i+size;
14 }
```

FIGURE 3.27 – Benchmark *kernel* OpenCL qui transfère les données de la mémoire DRAM vers les registres

Nous avons donc conduit une caractérisation exhaustive en variant le nombre de *work-items* exécutés ainsi que leur organisation en *work-groups*, afin de tester l'impact de la taille des work-group sur les débits.

Les *work-groups* influencent le comportement du compilateur, quand la taille du *work-group* est grande le nombre d'accès concurrents devient petit, et quand la taille des *work-groups* est trop petite les unités d'exécution sont sous exploitées. Les figures : fig 3.28, fig 3.29 et fig 3.30 sont les débits obtenus avec une taille de *work-group* faible. En revanche, la figure 3.33 correspond à une taille importante de *work-groups*. Les autres figures sont les tailles médianes, pour lesquelles nous avons mesuré les meilleurs résultats, en particulier le débit maximal que nous pouvons espérer au final en utilisant tous les unités d'exécution EU du GPU est de 12,5 GB/s, comme le montre la figure 3.31.

Nous avons aussi par ailleurs conduit des tests en écriture, c'est-à-dire les transferts des registres vers la mémoire DRAM (mémoire globale). Sans donner le détail des résultats, ces tests ont donné des résultats comparables à ceux en lecture.

Le meilleur débit semble donc correspondre pour nos architectures *Intel Core* à la valeur 8 pour le *work-group-size*.

Nous avons mis en évidence l'importance de ce paramètre, à savoir la taille du *work-group* (ou *work-group-size*) sur les performances. Ce paramètre a été implicite-

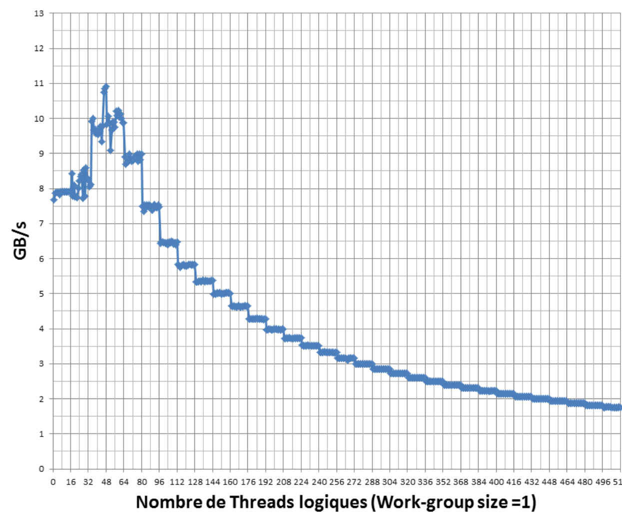


FIGURE 3.28 – Mise à l'échelle des débit Registres DRAM Work-group size =1

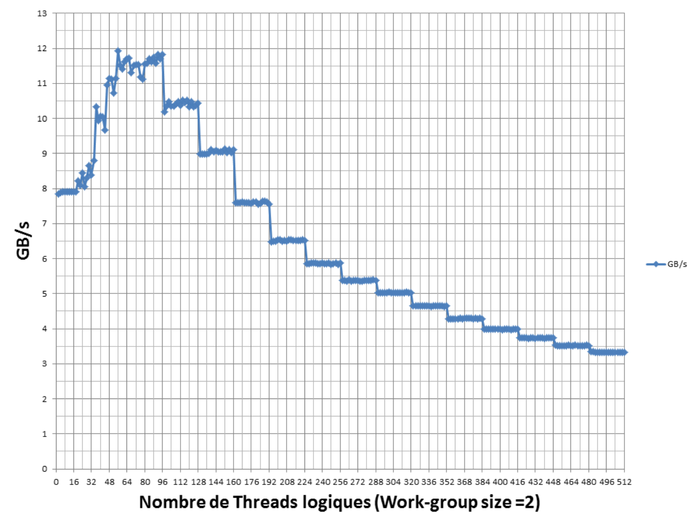


FIGURE 3.29 – Mise à l'échelle des débit Registres DRAM Work-group size =2

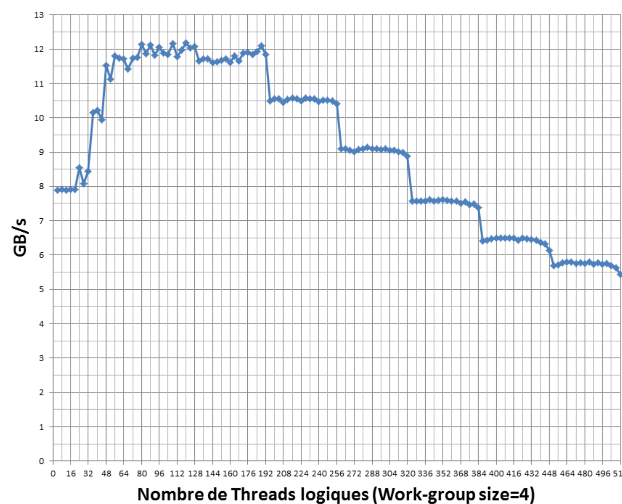


FIGURE 3.30 – Mise à l'échelle des débit Registres DRAM Work-group size =4

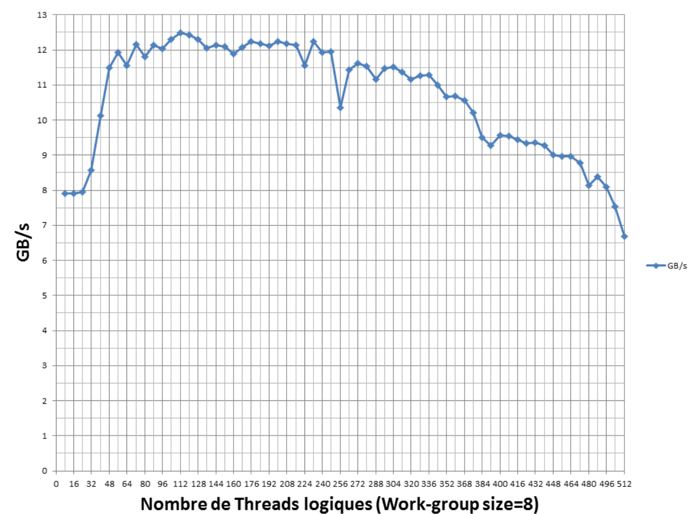


FIGURE 3.31 – Mise à l'échelle des débit Registres DRAM Work-group size =8

ment exploité dans le chapitre 4 afin d'adapter notre algorithme FFT à l'architecture GPU sous-jacente.

3.5.2.2 Calcul et taille du *work-group*

Comme vu précédemment, la taille des *work-groups* est un facteur influant sur les débits des transferts, ceci est aussi vrai pour les calculs.

Nous utiliserons le *benchmark* (3.5.1 figure 3.24) afin de caractériser l'influence du choix de la taille du *work-group* sur les performances de calcul.

La figure 3.34 montre que l'utilisation de 16 comme taille du *work-group* est a priori plus optimale que 8, ce qui garantit une utilisation efficace des unités de calculs. Nous constatons que 32, 64 ou même 128 sont aussi des tailles qui maximisent l'occupation des EU. Cependant, l'utilisation de tailles de *work-group* trop grandes

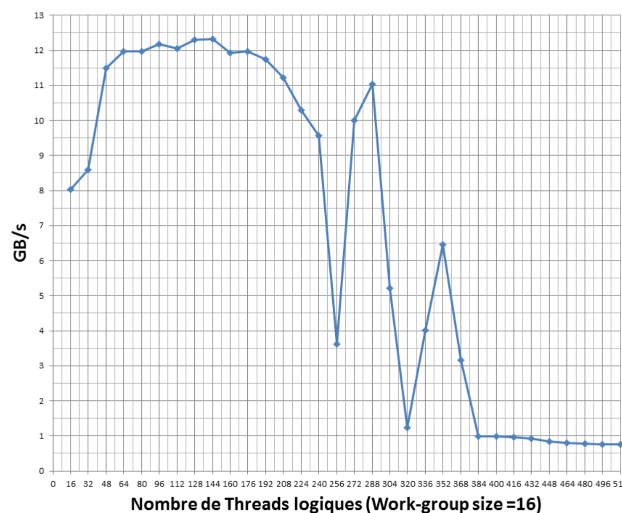


FIGURE 3.32 – Mise à l'échelle des débit Registres DRAM Work-group size =16

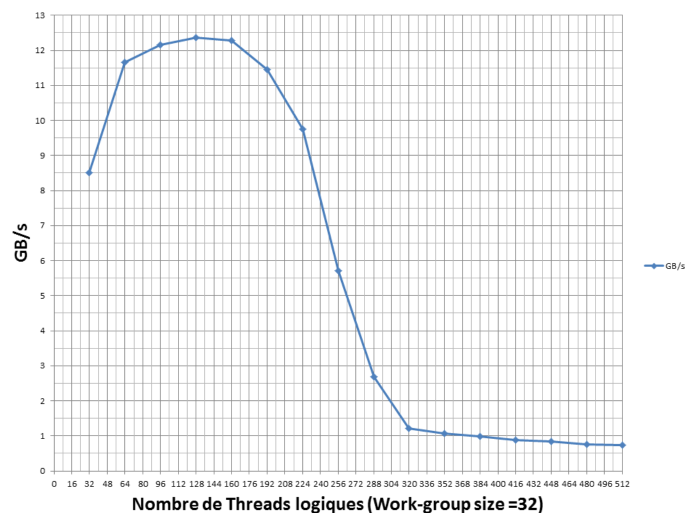


FIGURE 3.33 – Mise à l'échelle des débit Registres DRAM Work-group size =32

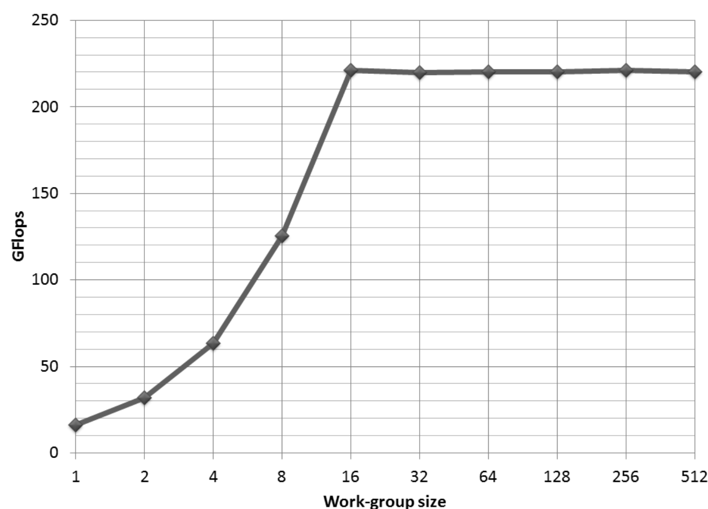


FIGURE 3.34 – Taille optimale du work-group

(c'est à dire > 128) nuit aux transferts de donnés.

Bilan et discussion

Les GPU sont une cible adaptée à notre cas d'utilisation. Nous ne considérerons pas dans cette thèse le cas des GPU externes, qui présentent deux handicaps majeurs, à savoir la consommation électrique et l'interconnexion avec le processeur hôte par le biais d'un bus PCI express plus lent et séquentiel. Mais notre cible chez Kontron, le processeur *Intel Core Haswell*, offre des GPU internes.

L'enjeu majeur est d'exploiter la puissance de calcul disponible dans les CPU modernes de l'architecture *Intel Core*. L'utilisation du standard OpenCL avec le modèle SIMT nous permettra d'écrire nos algorithmes de façons portable et efficace sur ce

type d'architecture.

Dans la suite de cette thèse nous allons étudier comment nous pouvons adapter de manière efficace notre algorithme FFT à la fois sur CPU et sur GPU en s'appuyant respectivement sur les deux modèles de programmation correspondants, SIMD et SIMT.

Chapitre 4

Adaptation de la FFT sur l'architecture

Motivations

Dans ce chapitre nous traitons le problème d'adapter au mieux les calculs de FFT vus au chapitre 2 sur l'architecture parallèle CPU-GPU des processeurs *Intel Core* décrite au chapitre 3. En raison des besoins de la société Kontron, nous nous sommes attachés à considérer des fonctions FFT de taille 1024, utilisées dans le cas de l'application radar. Cependant nous commenterons brièvement en fin de chapitre l'adaptation à des tailles différentes (2048, 4096,...).

L'objectif ici n'est **pas** de mobiliser le maximum de ressources matérielles pour le calcul d'une FFT unique, mais plutôt de trouver le meilleur ajustement « unitaire » entre une FFT et un cœur (tous deux élémentaires), afin de pouvoir calculer ensuite de nombreuses FFT en parallèle sur tous les cœurs disponibles, à partir de cet ajustement de base, comme le réclame l'application radar ou d'autres.

Les processeurs CPU/SIMD et GPU/SIMT possèdent des mécanismes parallèles de micro-architecture assez voisins. La situation idéale aurait alors été qu'une même version du code puisse être applicable aux deux cibles de compilation. Cela aurait induit des gains évidents

en productivité et maintenance de code (surtout sachant que les versions futures des processeurs *Intel* maintiendront une certaine stabilité architecturale, avec principalement des extensions en dimension des composants). Pour des raisons factuelles dues aux compilateurs courants cela n'a pu être le cas. En particulier, le code SIMD est actuellement *« scalarisé »* par le compilateur OpenCL GPU, alors que le code SIMT est lui mal vectorisé sur CPU. Nous reviendrons plus tard sur ces aspects qui nous ont contraint à considérer indépendamment les deux implémentations.

Dans un premier temps nous proposons une implémentation de nature SIMD sur un processeur GPU. Nous constatons alors un certain nombre de problèmes et de pénalités en performances. Celles-ci sont liées à l'implémentation du compilateur OpenCL sur ces processeurs. Nous expliquons ces phénomènes qui nous conduisent à considérer ensuite une formulation axée sur un modèle étendu SIMT pour l'ajustement de la FFT sur un processeur GPU. À nouveau, nous conduisons les expérimentations en performances qui se révèlent cette fois-ci très intéressantes. On notera que *Intel* ne propose pas à ce jour de bibliothèque FFT pour ce GPU.

Dans un second temps nous considérons l'implémentation « a priori naturelle » de nature SIMD sur un cœur CPU, et nous étudions les performances et leurs va-

riations induites par les variantes algorithmiques. Notre solution se révèle alors très compétitive, et en particulier plus efficace que les bibliothèques natives fournies par *Intel*.

Enfin, nous testons sur CPU la formulation orientée SIMT auparavant adoptée pour les GPU. Notre campagne d'expérimentation révèle une certaine dégradation des performances par rapport à la version SIMD, mais qui peut rester acceptable dans de nombreux cas au vu des autres critères de qualité mentionnés plus haut. Nous discutons ces aspects en conclusion.

4.1 Adaptation SIMD puis SIMT sur un bloc élémentaire GPU

Nous commençons notre étude par l'implémentation de la FFT sur UN cœur GPU (on rappelle l'absence de bibliothèque native sur GPU *Intel Core*).

4.1.1 Démarche d'adaptation sur GPU

Bien choisir la variante de l'algorithme est une étape clef dans l'approche d'adaptation. Ici nous avons fait le choix d'utiliser l'algorithme de *Stockham* à entrelacement en fréquence et l'approche *mixed-radix* comme base pour notre travail de parallélisation. Cet algorithme est généralement adapté à la parallélisation [11] et en particulier aux architectures GPU [28].

Les arguments en faveur de ce choix sont les suivants :

— Le nombre d'échantillons ou taille du signal à traiter. Dans notre cas, pour des signaux numériques en traitement radar, les tailles sont le plus souvent des puissances de deux, qui sont plus simples à traiter avec des algorithmes rapides conçus en majorité pour ces tailles-là.

— La structure de l'algorithme. Un algorithme à entrelacement en fréquence prend en entrée les échantillons dans l'ordre naturel. De la même manière, un algorithme *mixed-radix* est une généralisation de l'utilisation des classes [29] radix-2 et radix-4, voire radix-8. ce choix nous permettra de tester le radix le mieux adapté.

— Le type de ré-ordonnancement de l'algorithme : ici on s'affranchit de l'étape explicite du *bit-reverse* avec un algorithme de type *auto sort*.

Une fois la variante algorithmique choisie, il reste à l'ajuster au mieux à une structure architecturale donnée, en maximisant l'adéquation entre le parallélisme potentiel de l'application (qui a guidé le choix algorithmique), et le parallélisme réel de l'architecture. Dans notre cas nous choisissons de « mapper » le calcul d'une FFT sur un seul cœur GPU pour bénéficier de la localité en mémoire et permettre ensuite d'effectuer en parallèle autant de FFT qu'il y a de cœurs disponibles. Ce mapping dépendra donc évidement de la taille des ressources sur le cœur de calcul.

Il reste donc à définir les tailles appropriées pour un certain nombre de paramètres, qui définiront à leur tour le parallélisme exploitable dans la définition et l'écriture du *kernel* OpenCL implémentant la fonctionnalité. Nous détaillons ci-dessous les quatre paramètres sur lesquels nous agirons pour arriver à notre but, il sont aussi illustrés sur la figure 4.1 :

1. Le nombre de *work-items* nécessaires pour traiter une FFT : ce paramètre dépend de la taille de la FFT à traiter et aussi de la taille de la mémoire locale partagée. Ces paramètres sont fortement corrélés, du fait que le compilateur alloue en fonction du nombre de *work-items* et leur organisation en *work-group* la taille mémoire locale partagée nécessaire.
2. Le nombre de registres disponibles pour chaque *work-item* : ce paramètre dépend essentiellement de l'architecture du GPU. L'espace registres disponible détermine le nombre de points complexes à calculer par un *work-item* dans notre algorithme FFT.
3. La taille de la mémoire locale partagée (SLM : *Shared Local Memory*) : ce paramètre influe fortement sur le nombre de *work-items* à utiliser et aussi sur la taille totale de la FFT.
4. Le nombre de communications et d'échanges de données entre *work-items*. C'est l'un des paramètres clef de notre implémentation. Il faut à tout prix minimiser les échanges de données entre *work-items* (voire les supprimer).

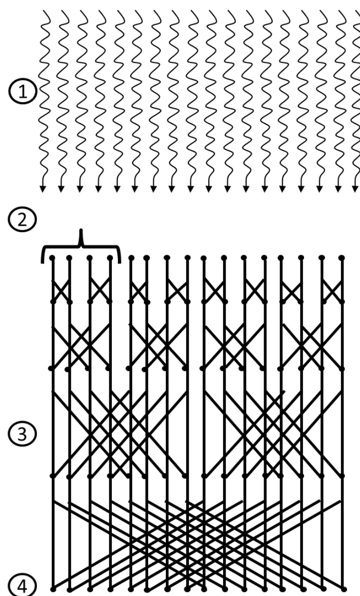


FIGURE 4.1 – Paramètres d'adaptation FFT

Les valeurs effectives pour ces paramètres sont calculées afin d'ajuster au mieux les dimensionnements. Nous décrivons dans la suite les choix et les calculs qui ont permis d'obtenir les meilleures performances possibles.

4.1.2 Cœur de calcul

Le cœur de calcul – ou bloc de base – d'une FFT est un papillon (voir 2.1.3). Pour traiter tous les points de la FFT, ce cœur sera itéré au sein d'un *work-item* en fonction du *radix* choisi.

Le papillon est implémenté en OpenCL avec l'instruction *fma(a,b,c) – fused multiply-add* – qui permet d'exécuter une multiplication et une addition en une seule étape sur le GPU (voir 3.1.1.1).

Toutes les unités d'exécution *EU* de l'architecture GPU *Intel Core* sont dotées d'unités SIMD capables d'exécuter efficacement 8 instructions FMA en parallèle sur 8 flottants de 32bits. L'architecture est parfaitement adaptée pour exécuter ces instructions, puisqu'elle est capable de charger depuis les registres de l'EU trois valeurs flottantes de 32bits par cycle d'horloge (c'est à dire 96bits/cycle) ce qui correspond aux trois opérandes nécessaires à une instruction FMA. Nous exploiterons aussi l'optimisation FMA décrite en section 2.1.3 qui réduit de 40 % le nombre d'opérations nécessaires (on passe alors de 10 opérations à 6). Le résultat OpenCL est (voir le papillon correspondant sur la figure 2.c) :

```
1        //im correspond à la partie imaginaire
2        //Re correspond à la partie réelle
3        //w correspond à la racine de l'unité
4        tmp1   =  fma(wimN, srcIm2, srcRe2);
5        tmp2   =  fma(wim, srcRe2, srcIm2);
6        dstRe1=  fma(tmp1, wre, srcRe1);
7        dstIm1=  fma(tmp2, wre, srcIm1);
8        dstRe2=  fma(tmp1, wreN, srcRe1);
9        dstIm2=  fma(tmp2, wreN, srcIm1);
```

Ce papillon prend ainsi 8 paramètres simple précision en entrée et en produit 4 en sortie. Ce qui fait un total de 12 flottants = 48Bytes. Nous soulignons aussi le besoin en variables temporaires, chaque papillon utilise 2 flottants = 8Bytes. Ceci porte le besoin mémoire pour notre cœur de calcul à 14 flottants = 56Bytes.

Sachant que dans une FFT la première passe met en jeu des *twiddle factors* triviaux, à savoir 1 pour la partie réelle et 0 pour la partie imaginaire, une première optimisation consiste alors à utiliser 4 opérations au lieux de 6 pour la première passe de la FFT. Nous obtenons ainsi le cœur de calcul trivial suivant pour la première passe de notre implémentation :

```
1        //im correspond à la partie imaginaire
2        //Re correspond à la partie réelle
3        dstRe2=srcRe1−srcRe2;
4        dstIm2=srcIm1−srcIm2;
5        dstRe1=srcRe1+srcRe2;
6        dstIm1=srcIm1+srcIm2;
```

Pour résumer, nous utiliserons pour la première passe le cœur de calcul trivial, et pour les passes suivantes le cœur de calcul de base constitué de 6 FMA.

4.1.3 Approche locale (« *in-register* »)

L'approche dite « *in-register* » consiste à calculer une FFT de taille N à l'aide d'une seule EU GPU. Notre objectif est d'effectuer tous les calculs dans les registres de l'EU afin d'avoir un maximum de localité et ainsi obtenir le maximum de performances crêtes, car on ne subit pas les pénalités des transferts mémoires. Ces derniers sont couteux non seulement en latence mais aussi en énergie.

Comme présenté au chapitre 3 (figure 3.20), chaque EU du GPU est dotée de deux unités de calcul. Chaque unité est de largeur 128bits (SIMD ALU 0 et SIMD ALU 1). Elles permettent d'exécuter chacune 8 opérations flottantes à chaque coup d'horloge, ce qui nous donne un total de 16 opérations flottantes simple précision par cycle. Ceci constitue la capacité de calcul disponible pour chaque EU.

Afin de masquer les latences induites par les opérations lentes (accès mémoire par exemple), les EU contiennent 8 Threads Hardware. Chaque thread détient un jeu de 128 registres généraux dédiés de 256bits. Au total, chaque EU contient 128×256bits $\times$ 8 Threads HW $= 32$KB.

Ceci constitue un espace registre important. Pour pouvoir effectuer notre approche, nous devons nous assurer que la totalité de cet espace mémoire est visible par chaque *work-item*, et qu'ils peuvent accéder à tous les registres disponibles au sein d'une unité d'exécution EU. Ceci a été vérifié en lançant un *work-item* sur une seule EU avec un calcul sur 32KB de données déclarée *__private* en OpenCL (voir le listing suivant).

```
__kernel void registerSize(__global float* res)
{
        __private float rg[8175]; //8175 registres
        __private float16 x=128; //16 registres
        __private int i=0;//1 registre =>8192 registres
        for(i=0;i<8175;i++)
        {
          rg[i]=0.2;
        }
        for(i=0;i<8175;i++)
        {
          rg[i%9]+=x.x+x.y*2+x.z;
        }
        //this confirm the size of private memory
        //=>32KB for one HW thread
        //to avoid the compiler from doing clever things
        res[0]=i+rg[8]+x.x;
}
```

On devrait donc disposer de 32KB de registres pour effectuer les calculs localement. Nous devons maintenant définir la taille N maximale de notre FFT pouvant être calculée avec ces 32KB.

4.1.3.1 Taille maximale de la FFT

Pour établir la taille maximale d'une FFT « *in-register* », on calcule la somme des besoins en mémoire. Pour une FFT de N points complexes nous avons besoin de $N \times 2$ registres simple précision pour stocker les données d'entrée (partie réelle et partie imaginaire pour chaque point). Nous avons aussi besoin d'un autre espace de même taille $N \times 2$ car nous utilisons l'algorithme de *Stockham* qui est un algorithme « *out-of-place* ». Cela porte l'espace mémoire à $N \times 2 \times 2$. Nous avons enfin besoin de stocker les coefficients *twiddle factors* dans les registres : ces derniers sont au total de $N \times 2$. Cependant, en utilisant le cœur de calcul basé sur des FMA, nous avons aussi besoin de stocker des *twiddle factors* négatifs. Cette contrainte est due à l'absence d'instruction de multiplication-soustraction fusionnée dans l'architecture. Ceci ramène finalement la taille mémoire nécessaire pour calculer une FFT complexe de taille N à $(N \times 2 \times 2) + (N \times 2 \times 2) = 8 \times N$.

Donc, pour trouver la taille de FFT maximale possible à calculer avec 32KB = 8192 float de registres, il suffit de résoudre $8192 = 8 \times N$. La taille maximale de FFT que nous pouvons calculer dans les registres d'une EU GPU est donc $N = 1024$. Cette

taille est aussi idéale pour l'application Radar de Kontron qui traite des données de taille 1024 points complexes.

Cependant, pour calculer un papillon FFT nous avons aussi besoin de deux registres temporaires. Ceci pose problème si nous ne souhaitons pas réduire la taille de notre FFT. Nous allons voir dans la section suivante comment optimiser l'utilisation de notre espace mémoire.

4.1.3.2 Optimisation de l'espace registres

Pour résoudre le problème d'adaptation de l'espace registres à notre besoin, nous nous sommes attelé à optimiser les ressources disponibles. La figure 4.2 illustre l'espace mémoire utilisé par notre implémentation (avant et après optimisation). Pour calculer une FFT de 1024 points complexes, nous avons besoin de deux tableaux T1 et T2, pour accueillir les échantillons sources, soit $8\text{KB} \times 2 = 16\text{KB}$ au total. Cet espace ne peut bien sûr pas être réduit.

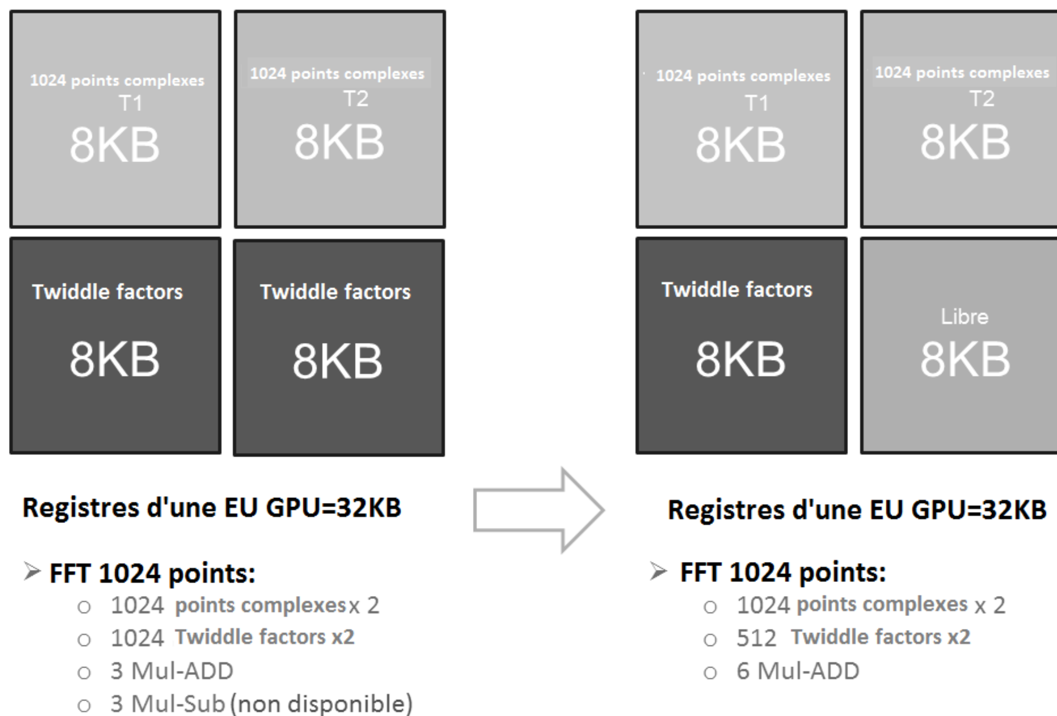


FIGURE 4.2 – Adaptation FFT 1K sur GPU au niveau des registres

Si on stocke naïvement tous les *twiddle factors* (valeurs positives et valeurs négatives) nécessaires pour notre algorithme, alors on a besoin à nouveau de 16KB de mémoire : au total on remplit les 32KB de registres de l'EU. Dans ce cas, il n'y a plus d'espace disponible pour les données temporaires nécessaires au calcul et cela entraine des débordements de registres – *register spilling* – qui vont nuire aux performances de notre implémentation. Afin de palier à ce problème, on exploite le fait que les *twiddle factors* correspondent aux racines n-ièmes de l'unité. Leurs parties réelles et imaginaires correspondent aux fonctions périodiques cosinus et sinus de période 2π.

Ainsi, nous pouvons stocker uniquement le quart de la période des *twiddle factors* et les autres valeurs seront déduites facilement à partir de ce quart. Cette manœuvre libère ainsi 12KB d'espace registres. Cependant, puisque le jeu d'instructions du GPU Intel ne posséde pas l'instruction multiplication-soustraction fusionnée, les opération de soustraction devront être encodées dans le signe des *twiddle factors*. Nous devons enregistrer deux versions de chaque *twiddle factor* : une avec le signe $(+)$ et une autre avec le signe $(-)$. Au final, nous devons donc stocker 8KB de *twiddle factors*, ce qui laisse 8KB de registres de libre pour accueillir les variables temporaires qui n'occupent que 2×16 papillons $= 32 => 128$B. Nous avons maintenant tout l'espace nécessaire pour effectuer notre calcul sans débordement de registres.

4.1.3.3 SIMD sur GPU en OpenCL

Notre *kernel* FFT consiste à calculer toute la FFT de 1024 points complexes sur une seule EU. Afin d'avoir un maximum de localité, nous avons lancé un seul *work-item* sur le GPU. Nous avons écrit notre algorithme en utilisant les types vectoriels OpenCL, afin d'exploiter toute la largeur de l'unité SIMD de l'EU. Nous chargeons au début du calcul les *twiddle factors* ainsi que les échantillons source dans les registres. Nous déroulons ensuite notre algorithme FFT vectoriel. Le listing suivant illustre la structure du *kernel* OpenCL correspondant.

```
1  __kernel __attribute__((vec_type_hint(float16)))
2  void FFT1024_16(__global float16* src1_Re,
3                          __global float16* src1_Im)
4  {
5          //twiddle factors
6          __private float2 twiddle[1024]={.......};
7          //allocate memory for src/dst
8          __private float16 src_mem_re[64];
9          .........
10         //fill registers from Global memory
11         for(i=0;i<64;++i)
12         {
13         src_mem_re[i]=src1_Re[i];src_mem_im[i]=src1_Im[i];
14         }
15         .........
16         //Pointers and temp. variables
17         __private float16 *src_re = src_mem_re;
18         .........
19         for(i=0;i<32;++i)
20         {
21         //first stage Compute
22         src_re[index1]= dst_re[index1]-dst_re[index2];
23         src_re[index2]= dst_re[index1]+dst_re[index2];
24         src_im[index1]= dst_im[index1]-dst_im[index2];
25         src_im[index2]= dst_im[index1]+dst_im[index2];
26         index1++;index2++;
27         }
28         //Intermediate Stages (stage two)
29         .........
30         //twiddle factor load
31         .........
32         for(i=0;i<16;++i)
33         {
```

```
34          .........
35          //compute
36          vtmp1=fma(dst_re[index3],vWreN,dst_im[index3]);
37          vtmp2=fma(dst_im[index3],vWre,dst_re[index3]);
38          src_re[index4]=fma(vWim,vtmp1,dst_re[index4]);
39          src_im[index4]=fma(vWimN,vtmp2,dst_im[index4]);
40          src_re[index2]=fma(vWimN,vtmp1,dst_re[index4]);
41          src_im[index2]=fma(vWim,vtmp2,dst_im[index4]);
42          .........
43          }
44          .........
45          //write to Global memory
46          .........
47  }
```

Cette approche « *in-register* » devrait en théorie produire des résultats quasi optimaux du fait de la localité des calculs dans les registres. Cependant, les performances obtenues se sont révélées décevantes, ne dépassant pas les 2GFlops FFT.

Après une analyse approfondie avec l'équipe d'experts GPU d'Intel, nous nous sommes rendu compte qu'en lançant 1 *work-item* pour traiter toute la FFT, les calculs se faisaient sur 1 ALU scalaire uniquement, ce qui revenait à n'utiliser que 1/8 de la capacité de calcul de l'unité d'exécution. Ceci est du au compilateur Intel qui « scalarize » le code vectoriel : il exécute l'instruction voulue au départ vectorielle en séquentiel sur une partie seulement de l'unité SIMD (voir figure 4.3), sans exploiter toute la capacité de calcul de l'EU.

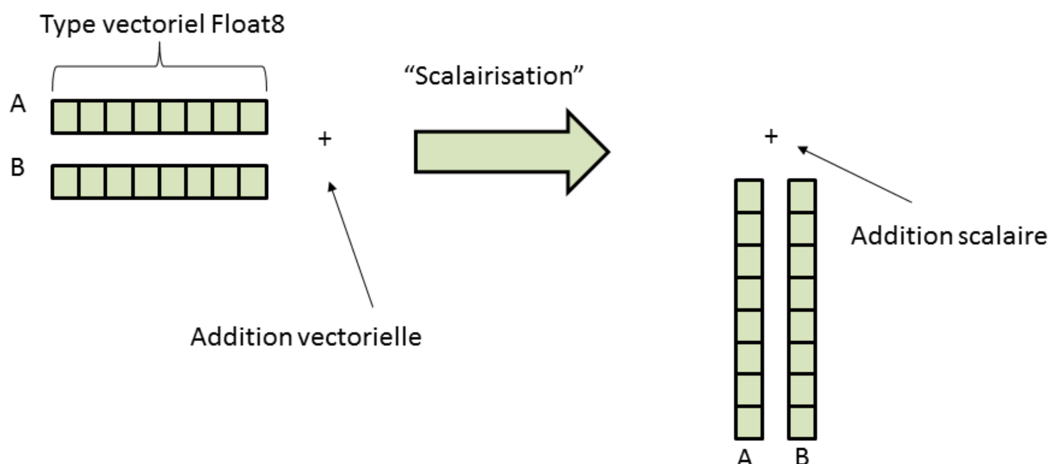


FIGURE 4.3 – Scalairisation d'une opération d'addition vectorielle

Après avoir constaté cette limitation, nous avons essayé de lancer 8 *work-items* afin d'exploiter toute la largeur en écrivant notre code de manière scalaire pour le rendre compatible avec le compilateur OpenCL d'Intel. Un autre problème est alors remonté à la surface : même si le hardware permet d'échanger des données entre *work-items* dans les registres, l'implémentation faite par Intel interdisait cette option. Nous notons aussi que sur recommandation de notre part, l'équipe du compilateur GPU d'Intel a accepté d'ajouter une extension au compilateur pour la prendre en charge. Les travaux présentés dans cette thèse ne portent pas sur l'utilisation de cette

extension récente apparue en janvier 2015.

Nous avons conclu que nous devrions alors utiliser une approche plus appropriées à la philosophie GPU / Compilateur OpenCL d'Intel à savoir l'approche SIMT.

4.1.4 Approche SIMT

Les architectures GPU sont des architectures optimisées pour le débit de calcul. Pour en tirer parti, il faut maintenir un très grand nombre de *threads* hardware occupées en parallèle. L'approche SIMT consiste à avoir des *work-items* en parallèle qui doivent exécuter le même *kernel* sur des données différentes. Dans notre cas, chaque *work-item* doit calculer une partie de la FFT sur des échantillons différents.

L'objectif ici est de trouver comment décomposer une FFT de taille N en *work-items*, afin d'exploiter au mieux le parallélisme fourni par les GPU *Intel Core*. Pour les mêmes raisons que dans l'approche précédente, notre but est d'exécuter une seule FFT sur une EU.

4.1.4.1 Adaptation au registres

Si on décompose de manière naïve l'algorithme FFT, on calculera un papillon radix-2 sur un *work-item* (voir figure 4.4). Nous aurons alors besoin de $N/2$ *work-items* en parallèle pour calculer une FFT de taille N.

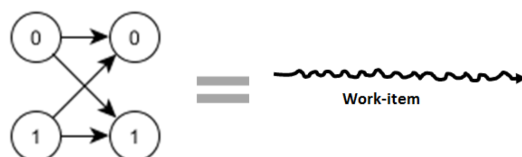


FIGURE 4.4 – Mapping FFT radix-2 sur 1 thread logique ou *work-item*

Chaque *work-item* calculera ainsi une FFT de taille $N_1 = 2$, ce qui correspond à un papillon FFT. En se basant sur le cœur de calcul défini plus haut en section 4.1.2, l'espace registres nécessaire par ce papillon est de 14 registres 32bits. Ceci revient à $14 \times \frac{N}{N_1}$ registres 32bits pour calculer toute la FFT de taille N.

Nous avons vu précédemment que la plus grande FFT que nous pouvons calculer avec 32KB de registres disponibles pour une EU est une FFT de 1024 points complexes. Dans notre cas, les 32KB de registres seront divisés entre les *work-items*, nous allons voir comment bien gérer cet espace registres de l'EU qui en OpenCL correspond à la mémoire privée.

Nos expérimentations du chapitre 3 sur l'architecture nous ont permis d'éviter quelques erreurs d'implémentation sur l'utilisation de cette mémoire privée GPU. En réalité, cet espace mémoire, même s'il est explicitement typé à l'aide du mot clef « __private » en OpenCL, est alloué par le compilateur en priorité si possible vers les registres, sinon il est alloué dans la mémoire globale, à savoir la DRAM. Un mauvais choix de taille se traduira par une allocation DRAM et des accès mémoires globaux très couteux en énergie et en latence. Ce paramètre est d'autant plus important que

les outils fournis par Intel ne donnent aucune indication sur l'emplacement physique où sont allouées les données privées.

Effectuer une FFT 1024 points à l'aide de 512 *work-item* correspond au parallélisme maximum pour cet algorithme. Cependant, l'occupation des *work-items* reste faible. Nous chercherons alors à trouver la taille des données idéale à traiter par chaque *work-item*.

Si on note p le nombre de points à traiter par un *work-item*, et r l'espace mémoire disponible pour chaque *work-item*, nous devons donc avoir l'inégalité 4.1, afin que le compilateur alloue efficacement les registres du GPU (sans débordement).

$$p < r \tag{4.1}$$

Le compilateur Intel définit deux modes d'exécution, SIMD8 et SIMD16. On a vu section 3.4.2 que le mode SIMD8 correspond à un mapping 1 à 1 entre un *« lane »* SIMD et un *work-item*. Le mode SIMD16 correspond à l'exploitation de l'unité ALU SIMD en *double-pumped*, c'est-à-dire que nous pouvons mapper 16 *work-items* sur l'unité SIMD qui est de largeur 8 à la base. Nous notons v le mode de compilation SIMD choisi, $v \in \{8, 16\}$.

Nous rappelons aussi que chaque EU possède 8 Threads Hardware. Nous notons alors th le nombre de *Threads hardware*. Chaque EU a une quantité de registres dédiés notée ici q.

Notre espace registres r disponible pour chaque *work-item* s'écrit alors sous la forme :

$$r = q/(v \times th) \tag{4.2}$$

Donc l'inégalité 4.1 devient :

$$p < q/(v \times th) \tag{4.3}$$

Dans notre cas $p < \frac{32\text{KB}}{16\times 8}$, donc nous aurons $p < 64$ points flottants simple précision.

En réalité, pour calculer une FFT de N points complexes, nous avons besoins de $2 \times N$ valeurs. Aussi, si nous utilisons l'algorithme de *Stockham* qui est *« out-of-place »* nous aurons besoin du double de l'espace. Ce qui veut dire $2 \times 2 \times N$. Donc notre inégalité devient $p < (r/4)$, dans notre cas alors $p < 16$ points complexes flottants simple précision. Nous aurons aussi besoin de stocker au minimum 1 *twiddle factor* ce qui rend le nombre de points forcément inférieur à 16.

Les choix qui se posent à nous par conséquence sont 8 (voir figure 4.6), 4 (voir figure 4.5) ou 2 points complexes.

Le nombre de points à traiter par chaque *work-item* définit le nombre de *work-items* en parallèle qu'il faut utiliser pour arriver à calculer une FFT de taille N. Le nombre de *work-items* à utiliser noté wi est : $wi = \frac{N}{p}$.

Dans notre cas, pour calculer une FFT 1024 points, nous aurons besoin de 512, 256 ou de 128 *work-items* pour des valeurs respectives de p de 2, 4 et 8. Tous ces *work-items* doivent s'échanger leurs résultats, ce qui n'est possible qu'à travers la mémoire partagée. Nous étudions dans la section suivante comment nous adaptons notre *kernel* pour exploiter efficacement cette mémoire.

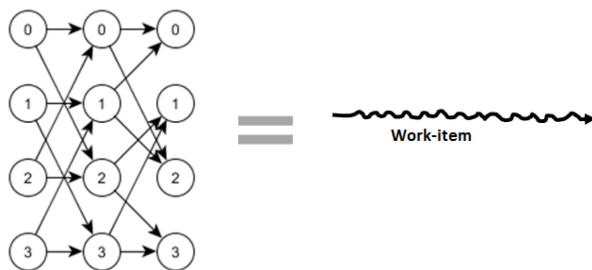


FIGURE 4.5 – Mapping FFT radix-4 sur 1 thread logique ou *work-item*

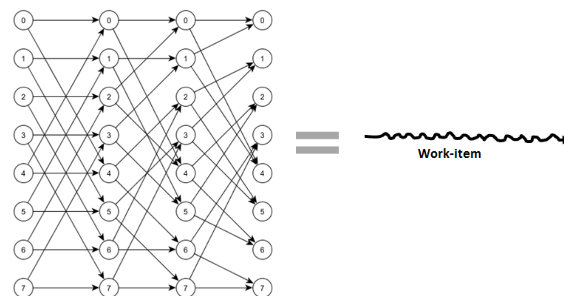


FIGURE 4.6 – Mapping FFT radix-8 sur 1 thread logique ou *work-item*

4.1.4.2 Adaptation à la mémoire partagée

Nous avons vu dans le chapitre 2 que pour calculer une FFT de 1024 points complexes en radix-2 nous avons besoin de 10 étages de calculs ($2^{10} = 1024$). Chaque étage consistera en $1024/2 = 512$ *work-items* en parallèle. Chaque *work-item* calculera ainsi un papillon sur deux valeurs complexes.

Entre chaque étage, les *work-items* échangeront les résultats de leurs papillons respectifs. OpenCL définit un espace mémoire « __local » accessible par tous les *work-items* d'un même *work-group*. Cette mémoire est allouée dans le cache L3 (voir 3.4.3). Nous rappelons que cette zone mémoire est de type « *scratchpad memory* » donc elle est explicitement gérée.

Chaque *work-item* écrira 4 valeurs flottantes (le résultat du papillon de l'étage $i-1$) et lira 4 autres valeurs flottantes (les valeurs d'entrée pour calculer le papillon de l'étage i).

Après écriture du résultat dans la mémoire partagée (SLM), les *work-items* doivent attendre que tous les autres *work-items* du même *work-group* aient fini d'écrire. Nous devons placer des barrières de synchronisation après chaque écriture dans la mémoire partagée (SLM) pour s'assurer de ne pas avoir d'accès mémoire en lecture et écriture concurrents.

Le code OpenCL suivant écrit le résultat du papillon dans la mémoire partagée (SLM) et place ensuite une barrière de synchronisation.

```
1  //écriture de deux points complexes dans la mémoire partagée
2        SLM[out1] = (float2)(dstRe1,dstIm1);
3        SLM[out2] = (float2)(dstRe2,dstIm2);
4  //Barrier de synchronisation
5        barrier(CLK_LOCAL_MEM_FENCE);
```

L'espace mémoire partagée dont nous avons besoin est équivalent aux données d'entrée d'une FFT de taille N. Cet espace correspond à $N \times 2$ valeurs simple précision pour une FFT complexe. Dans notre cas nous utiliserons $1024 \times 2 = 2048 \times 8$B => 8KB d'espace mémoire partagée pour chaque EU. Sur la figure 4.7 nous constatons qu'avec une mémoire partagée de taille 128KB sur notre architecture, nous pouvons traiter 16 FFT de 1024 points complexes en parallèle 16×8KB = 128KB.

Nous formalisons alors la taille FFT adéquate par rapport à l'espace mémoire partagée disponible par la formule 4.4. Pour une FFT complexe simple précision de taille N, nous avons besoin de 8Bytes pour chaque point. Nous notons Sh la taille de la mé-

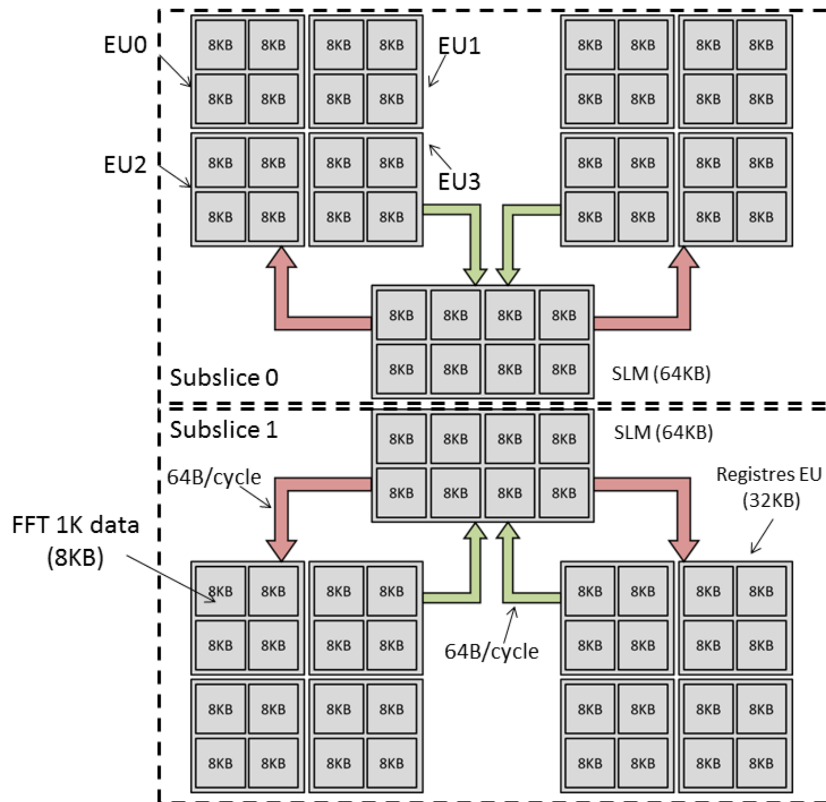


FIGURE 4.7 – Utilisation de la mémoire partagée SLM pour échanger 8KB de données FFT 1024 points complexes

moire partagée disponible. Nous notons aussi nEU le nombre de EU disponibles.

$$N = Sh/(8 \times nEU) \tag{4.4}$$

Nous constatons ici que la taille de FFT 1024 points est idéale pour notre architecture pour deux considérations principales : la taille des registres et l'utilisation de la mémoire partagée.

Nous signalons que dès qu'on utilise la mémoire partagée pour échanger les données entre *work-items* nous sommes obligés de placer des barrières de synchronisation. Comme vu en 3.4 nous n'avons que 32 barrières de synchronisation fournies par l'architecture. Nous devons donc les utiliser avec parcimonie.

Nous notons aussi que nous avons fait le choix d'allouer les *twiddle factors* dans la mémoire dite « __constant » . Cette mémoire est accessible par tous les *work-items* et aussi par tous les *work-groups*. Nous avons fait ce choix car cette mémoire bénéficie de la hiérarchie du cache du GPU ce qui lui permet d'avoir une latence de lecture très faible (meilleure que celle de la mémoire « __global »).

Avec 512 *work-items* pour calculer une FFT 1024 points complexes en radix-2 nous avons besoin de 8 phases d'échanges de données entre les *work items* de notre *work-group*. La figure 4.8 illustre l'échange de données nécessaire pour notre implémentation FFT 1024 points complexes. Ceci nécessite alors 8 barrières de synchronisation par *work-group*, ainsi que 8 écritures et 8 lectures de la mémoire partagée.

Deux handicaps majeurs pénalisent cette implémentation : le premier est le ratio

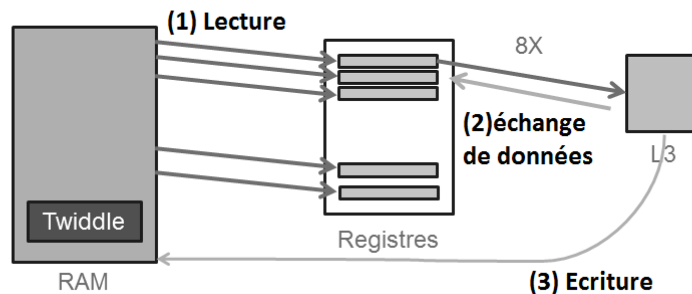


FIGURE 4.8 – Échange de données implémentation FFT 1K (radix-2)

entre les calculs et les communications pour un *work-item* (entre les registres et la mémoire partagée) ; le deuxième est l'utilisation inefficace de la mémoire partagée.

Nous définissons le ratio entre les calculs et les communications par le rapport entre les calculs utiles d'un *work-item* (le nombre d'opérations FMA) et les transferts mémoire en lecture et écriture entre les registres et la mémoire partagée. Ce ratio sera noté R_{cc}.

Nous notons cal le nombre d'opérations FMA de notre *kernel*, cette valeur ne changera pas, nous aurons alors $6 \times 10 \times 512 = 30720$ opérations. Nous notons aussi com le nombre de valeurs écrites et lues dans la mémoire partagée, ici 4 valeurs écrites et 4 valeurs lues par phase d'échange.

$$R_{cc} = cal/com \tag{4.5}$$

Donc d'après l'équation 4.5, notre ratio $R_{cc} = \frac{30720}{4\times8\times2\times512} = 0,93$. Ce ratio nous servira pour comparer nos différentes implémentations entre elles. Nous pouvons anticiper que nous avons beaucoup trop de communications par rapport aux calculs. Nous essayerons par la suite de minimiser l'impact des communications par rapport aux calculs.

Le deuxième handicap de cette implémentation cité plus haut, à savoir l'utilisation de la mémoire partagée, consiste dans le fait que le GPU Intel partage cette mémoire entre les différents *work-groups* de manière systématique. On note wg la taille de nos *work-groups* et Sh la taille de notre mémoire partagée. On note aussi v le mode SIMD utilisé (8 ou 16), th le nombre de threads hardware pour chaque EU et nEU le nombre de EU disponibles. La taille mémoire partagée disponible pour chaque *work-groups* est notée Sw.

$$Sw = sh \times wg/(v \times th \times nEU) \tag{4.6}$$

Dans notre cas nous aurons $Sw = \frac{128\text{KB}\times512}{(16\times8\times16)} = 32\text{KB}$ ce qui représente 4 fois plus que l'espace utile (8KB).

Malgré l'effort de parallélisation SIMT de notre algorithme, nous avons obtenue des performances FFT similaires à ceux de l'approche « in-register », en l'occurrence 2GFlops FFT. Cette performance assez moyenne nous servira de point de comparaison pour nos optimisations futures.

Il faut donc trouver d'autres alternatives pour améliorer le ratio R_{cc} et optimiser l'utilisation de la mémoire partagée.

Nous calculerons cette fois ci 4 points complexes par chaque *work-item*. Donc nous aurons $N/4$ *work-items* en parallèle qui traitent chacun un papillon de type radix-4.

Cette implémentation nécessite d'organiser nos calculs de sorte à avoir deux papillons qui s'exécutent de manière séquentielle par chaque *work-item*. Avec 4 points complexes nous pouvons calculer deux étages de la FFT avec les mêmes données. Nous avons alors pour une FFT de 1024 points, 256 *work-items* en parallèle. Nous avons aussi non plus 10 étages mais 5 étages de calcul radix-4. Ceci réduit les échanges de données entre *work-items*, car nous échangeons les résultats qu'après chaque étage radix-4. Nous réduisons ainsi par deux le nombre de lectures et d'écritures de et vers la mémoire L3 (SLM).

Notre ratio $R_{cc} = \frac{30720}{8 \times 4 \times 2 \times 256} = 1,87$ est ainsi amélioré.

Pour ce qui concerne l'utilisation de la mémoire partagée, nous obtenons alors $Sw = \frac{128\text{KB} \times 256}{(16 \times 8 \times 16)} = 16\text{KB}$, ce qui signifie que nous allouons 2 fois plus de mémoire partagée que nécessaire.

Notre expérimentation a montrée aussi un gains en performance net par rapport à l'implémentation radix-2. Nous avons obtenue ainsi une performance de 3,5 GFlops FFT.

De la même manière un papillon radix-8 calculé par un *work-item*, nous donne $N/8$ *work-items* en parallèle. Nous aurons ainsi dans notre cas 128 *work-items*.

Nous aurons alors 4 papillon calculés en séquentiel par chaque *work-item*.

Le ratio $R_{cc} = \frac{30720}{16 \times 3 \times 2 \times 128} = 2,5$ a aussi été amélioré, ainsi que l'utilisation de la mémoire partagée $Sw = \frac{128\text{KB} \times 128}{(16 \times 8 \times 16)} = 8\text{KB}$.

Radix-8 étant la limite pour l'architecture Intel GPU, nous adopterons cette solution et nous discuterons l'organisation des calculs dans la section suivante.

4.1.4.3 Organisation des calculs FFT

Nous savons qu'un radix-8 consiste à calculer une FFT de taille 8, ce qui correspond à 3 étages de calcul ($\log_2(8) = 3$). Donc, une FFT de taille N est calculée en $\log_2(N)/3$ étages radix-8. Dans notre cas, pour $N = 1024$, nous avons besoin de 3 étages radix-8 et un étage supplémentaire en radix-2.

Un calcul radix-8 correspond à 4 papillons FFT sur 3 étages, chaque papillon consiste à 6 FMA, ce qui nous donne au total $4 \times 6 \times 3 = 72$ opérations FMA.

Nos papillons s'exécutent alors de manière séquentielle par chaque *work-item* afin de calculer un bloc FFT radix-8 (voir figure 4.9). Nous n'utiliserons pas de boucle afin d'avoir un code régulier pour tous les *work-items*.

Chaque *work-item* calcule alors 3 étages de la FFT, sans recharger les données de la mémoire SLM. Il effectue ensuite un calcul d'indices (on peut parler aussi de permutation) afin d'écrire son résultat au bon endroit dans la mémoire partagée SLM. Une barrière de synchronisation est placée pour garantir la consistance des données, avant de lancer une lecture sur les données de la passe suivante. Nous rappelons que les barrières assurent que toutes les instructions avant la barrières soient exécutées avant d'entamer la suite des instructions.

La dernière passe nécessitera quand à elle l'enchainement de 4 papillons radix-2 afin de finir le calcul.

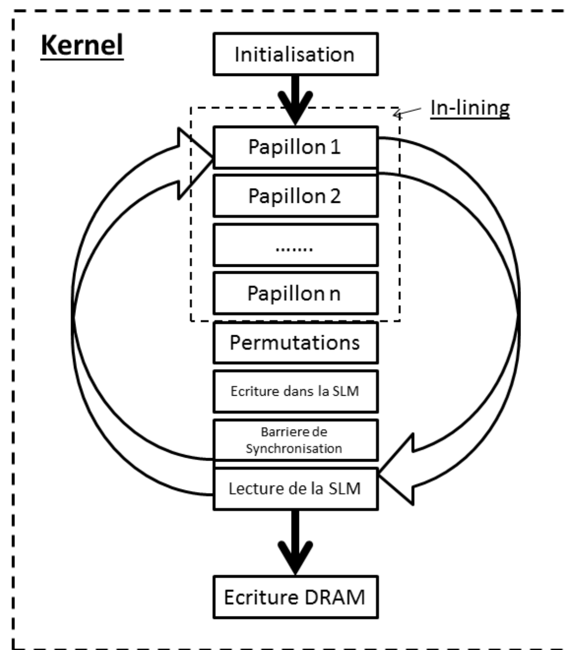


FIGURE 4.9 – Organisation des calculs FFT sur un *work-item*

La figure 4.10 schématise les mouvements de données induits par cette organisation. Nous notons que les accès vers la mémoire partagée sont aux nombre de 3, ce qui représente une réduction significative des communications par rapport aux autres implémentations. Nous notons aussi que ceci représente la limite basse du nombre de lectures/écritures vers la SLM.

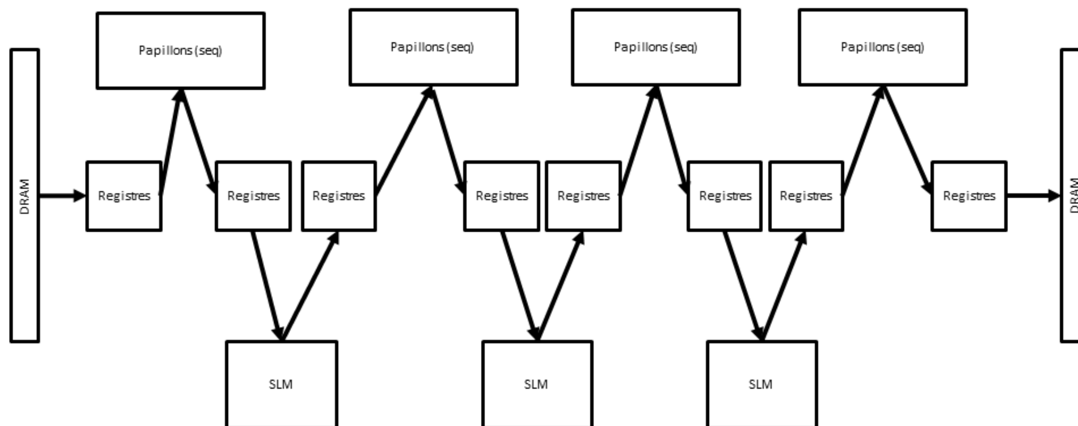


FIGURE 4.10 – Mouvement des données pour une FFT 1K points sur GPU

D'autres options d'organisation de notre algorithme s'offrent à nous. Afin de décomposer nos 10 passes de calcul radix-2, nous avons fait le choix de garder une localité de calcul au début et à la fin de notre FFT. Nous avons ainsi découpé nos 10 passes en des blocs de 3, 2, 2, 3. La figure 4.11 illustre alors notre implémentation FFT pour un *work-item*.

Pour calculer une FFT de 1024 points complexes nous lançons alors un *work-group* de 128 *work-items* ($128 \times 8 = 1024$). La figure 4.12 illustre le mapping de notre

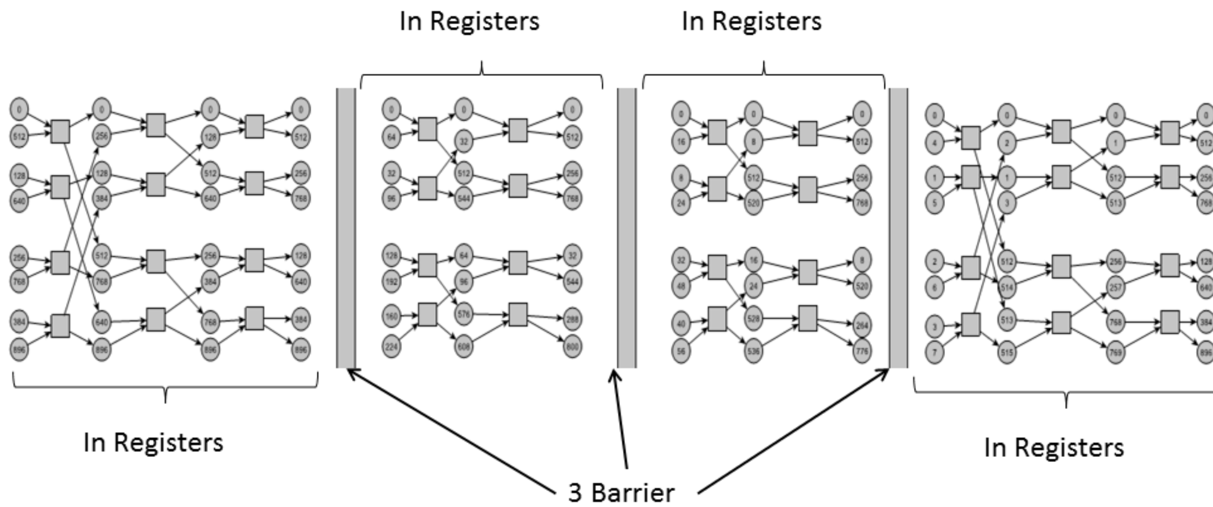


FIGURE 4.11 – Organisation de l'implémentation FFT 1K sur un *work-item*

implémentation sur 128 *work-items*. Chaque thread hardware exécutera alors 16 *work-items* en parallèle ($16 \times 8 = 128$). Ce mapping correspond à l'occupation totale d'une EU. Nous pouvons alors lancer autant de FFT en parallèle que d'EU disponibles.

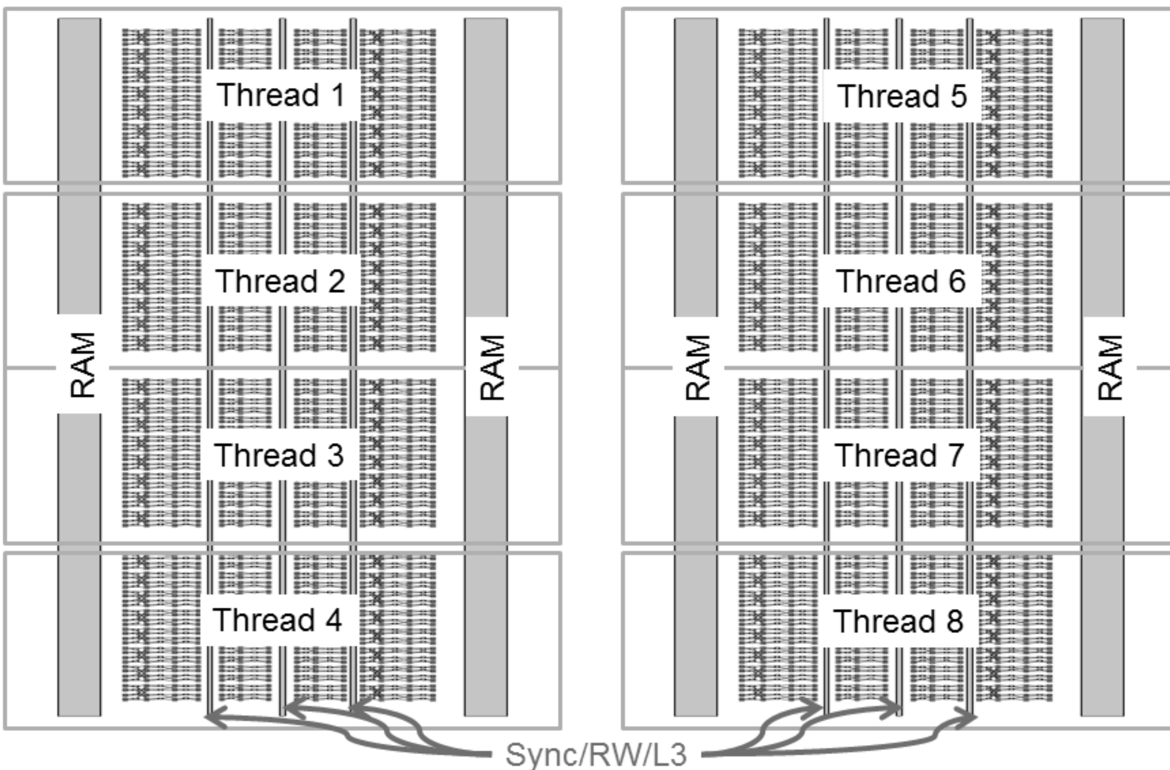


FIGURE 4.12 – Organisation de l'implémentation FFT 1024 sur 128 *work-items*, et leurs mapping sur les Threads Hardware de l'EU

Nous avons ainsi pu obtenir les très bonnes performances de 8GFlops FFT (inégalées jusqu'à présent), ceci grâce aux différentes optimisations successives : utilisation de l'instruction FMA, adaptation des tailles de registres et minimisation des échanges

de données.

L'utilisation d'algorithmes FFT de type radix-r nécessite que la taille de la FFT à traiter soit de taille $N = R^x$ avec $x = \log_R(N)$. Sachant que les tailles des signaux à traiter sont de tailles de type $N = 2^n$ (avec $n \in \mathbb{N}$) nous aurons des cas où il faudrait combiner plusieurs algorithmes pour arriver à finir le calcul. Dans notre cas, la FFT de taille 1024 points peut être traitée en 10 étages de calcul en radix-2. Cependant, avec un radix-8, nous ne pourrons traiter que les trois premiers étages en radix-8 et le calcul terminera avec une passe en radix-2. Ceci ne constitue pas la seule solution, différents arrangements sont possibles : radix-8 (3 passes), radix-4 (2 passes), radix-4 (2 passes), radix-8(3 passes) ou 3,3,2,2 ou 2,3,2,3 ...

Ce genre de combinaison est dite de type mixed-radix FFT et une question légitime qui se pose est quelle est la meilleure manière d'ordonner les passes pour calculer notre FFT de 1024 points ?

Nous tenterons de répondre à cette question dans la section suivante.

4.1.4.4 Approche adaptative FFT

Nous souhaitons trouver la meilleure manière pour ordonner les passes pour calculer notre FFT 1024 points. Ce qui se traduira prosaïquement dans notre cas à chercher le meilleur « découpage » de 10 étages en phases de 3 étages maximum. Dans la littérature FFT nous trouvons la notion de plan d'exécution[1, 30, 31]. Ce qui signifie que par exemple, dans le cas de FFTW, un plan d'exécution correspond à une suite de *« codelettes »* (des bouts de code très optimisés de façon unitaire) qui permet de calculer une FFT sur une architecture donnée. FFTW commence par générer toutes les combinaisons possibles pour arriver à un résultat, il les exécute ensuite sur l'architecture cible et essaye de converger vers la solution qui donne les meilleures performances. Une fois la solution trouvée, celle ci est stockée dans un plan d'exécution. Cette méthode donne souvent de bons résultats sur CPU, cependant il lui faut des heures voire des jours pour converger. Nous signalons aussi que FFTW est une bibliothèque qui existe seulement sur CPU.

Nous proposons une approche similaire dans le principe, cependant, notre méthode réduit l'espace d'exploration et réduit ainsi le temps pour converger vers la solution optimale.

Notre approche consiste à modéliser l'espace d'exploration (les différentes combinaisons de radix-2, radix-4 et radix-8) sous forme de graphe orienté. Ce graphe est doté d'un nœud source qui représente le début de la FFT et plusieurs nœuds destination qui représentent la fin de la FFT. Chaque arête représente un type de radix. Nous nous sommes restreint à trois types de radix, à savoir radix-2, radix-4 et radix-8. Chaque arête se voit attribuée un poids qui correspond au temps nécessaire en nanoseconde à l'exécution de ce nœud. Le but est de trouver le plus court chemin pour arriver à la fin de la FFT (nœuds en rouge)(voir figure 4.13). Nous utilisons l'algorithme de *Dijkstra*[32] pour trouver le plus court chemin.

Afin d'affecter les différents poids aux arêtes, nous devons tester un sous ensemble d'implémentations unitaires de type radix-r ($r \in \{2, 4, 8\}$). Pour notre cas, pour une FFT de 1024 points nous aurons 10 étages. Nos tests seront donc :

— 10 fois radix-2 en commençant par l'étage 0, l'étage 1 ... 9

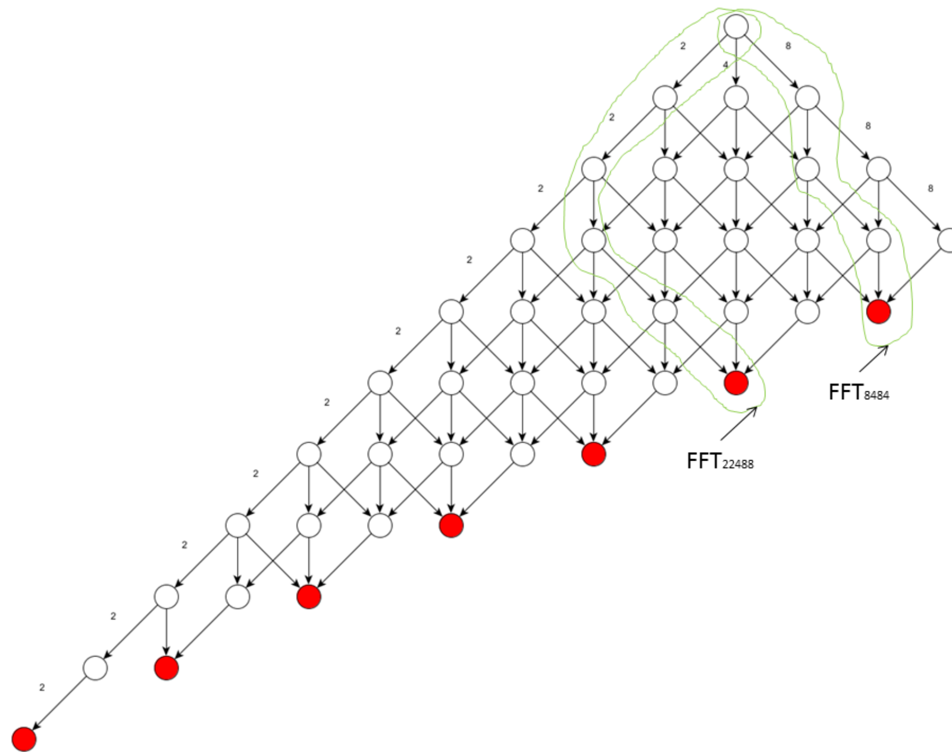


FIGURE 4.13 – Design Space FFT à explorer

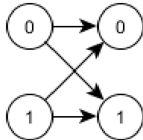


FIGURE 4.14 – Radix-2

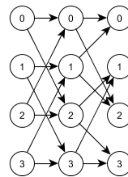


FIGURE 4.15 – Radix-4

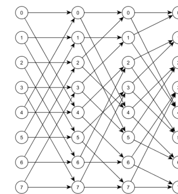


FIGURE 4.16 – Radix-8

— 9 fois radix-4 en commençant par l'étage 0, 1, ... 8
— 8 fois radix-8 en commençant par l'étage 0, 1, ... 7

Ce qui nous fait 27 tests sachant que la taille de l'espace d'exploration est 247. Ceci représente un gain de temps considérable par rapport aux autres approches.

le tableau 4.1 montre le nombre d'essais qu'il faut faire pour notre approche en fonction de la taille de la FFT.

Intuitivement, il semblerait que la meilleure combinaison est 8,8,8,2 (radix-8, radix-8, radix-8, radix-2). Nous noterons cette implémentation par $FFT_{8,8,8,2}$.

Nous l'avons expérimenté sur notre architecture cible *Intel Core GPU* et nous relevons les performances de chaque test en nanosecondes dans les tableaux suivants 4.2 4.3.

Notre approche démontre que la solution intuitive, à savoir $FFT_{8,8,8,2}$, n'est pas la meilleure combinaison possible. Nous obtenons ainsi que pour le GPU *Intel Ivy-Bridge* la meilleure combinaison est $FFT_{8,8,4,4}$. Cette dernière fournit un gain de performances de 10% par rapport à la solution intuitive $FFT_{8,8,8,2}$, ainsi que 39% par rapport à l'implémentation naïve.

$\frac{FFT_{8,8,8,2}}{FFT_{8,8,4,4}} = 1,10$ $\frac{FFT_{2,2,2,2,2,2,2,2,2,2}}{FFT_{8,8,4,4}} = 1,39$

Taille FFT	Nombre de mixed-radix	Nombre d'expérimentations
16	7	9
32	13	12
64	24	15
128	44	18
256	81	21
512	149	24
1024	247	27
2048	504	30
4096	927	33
8192	1705	36
16384	3136	39

TABLE 4.1 – Nombre d'implémentations possibles mixed-radix en fonction de la taille FFT

stage	radix 2	radix 4	radix 8
0	1600	3100	4135
1	2430	3660	4830
2	2600	3600	5520
3	2658	4002	5988
4	2560	4213	6480
5	2790	3910	7320
6	2600	4632	7896
7	2889	4510	7887
8	3512	5030	X
9	3913	X	X

TABLE 4.2 – Intel Ivy Bridge GPU tests (en ns)

stage	radix 2	radix 4	radix 8
0	1514	2813	3927
1	2295	3463	4569
2	2460	3407	5223
3	2513	3785	5666
4	2427	3985	6129
5	2640	3695	6930
6	2460	4385	7472
7	2737	4266	7445
8	3321	4240	X
9	3705	X	X

TABLE 4.3 – Intel Haswell GPU (en ns)

De manière équivalente, nous obtenons pour le GPU *Intel Haswell* que la $FFT_{8,8,4,4}$ est la solution optimale, cette dernière fournit les gains en performances suivants :

$\frac{FFT_{8,8,8,2}}{FFT_{4,8,8,4}} = 1,14$ $\frac{FFT_{2,2,2,2,2,2,2,2,2,2}}{FFT_{4,8,8,4}} = 1,43$

4.1.5 Synthèse de code

En fonction de nos résultats, le code à produire consistera à déplier (« inlining ») le nombre prescrit d'étages de la FFT pour chacun de nos composants (a priori parfois 2 et parfois 3 étages dans notre solution), en intégrant les permutations internes par changement direct des indices dans le code.

Ce qui donne :

```
__kernel void kvf_fft(int N,int d,__global float2* restrict SampleIn,
                              __global float2* restrict SampleOut)
{
        // shared local memory
        __local float2 SLM[1024]; //this is equivalent to 2048 Float
        ...
        //allocation of registers
        ...
        //Stage 1
        //precalculate idices
        ...
        read_From_Global_memory();
        //perform computation
        FFT8();
        write_To_shared_memory();
        //Barrier to synchronise all workGroup Threads before the next stage
        barrier(CLK_LOCAL_MEM_FENCE);
        //Stage 4
        //precalculate indices
        ...
        read_From_Shared_memory();
        FFT4();
        FFT4();
        write_To_shared_memory();
        barrier(CLK_LOCAL_MEM_FENCE);
        //Stage 6
        //precalculate indices
        ...
        read_From_Shared_memory();
        FFT4();
        FFT4();
        write_To_shared_memory();
        barrier(CLK_LOCAL_MEM_FENCE);
        //Stage 8
        //precalculate indices
        ...
        read_From_Shared_memory();
        FFT8();
        write_To_Global_memory();
}
```

4.1.6 Expérimentation et résultats pratiques

Afin de mesurer les performances de manière fiable et reproductible, nous lançons alors nos calculs plusieurs fois, et collectons les performances de chaque itération dans un tableau. Nous calculons ensuite la moyenne géométrique sur notre tableau afin d'obtenir notre performance finale en régime établi. Nous notons aussi

que le nombre d'itérations doit être suffisamment grand afin de garder une stabilité par rapport au temps. En plus, nous nous assurons qu'aucun autre processus système ne vient altérer nos mesures.

```
for(nbr_of_repeats)
{
        ret = clEnqueueNDRangeKernel(...  , &ev);
        clWaitForEvents(1,&ev);
        clGetEventProfilingInfo(ev,CL_PROFILING_COMMAND_START,  ...);
        clGetEventProfilingInfo(ev,CL_PROFILING_COMMAND_END,  ...);
        times[i] = end - start;
        ret = clReleaseEvent(ev);
}
//performance finale
Perf=Geometric_mean(times[]);
```

La figure 4.17 montre les performances obtenues pour notre implémentation FFT sur GPU Intel IvyBridge et Haswell. Sachant que le maximum théorique en terme de calculs flottant pour chaque EU est 16GFlops, notre implémentation FFT fournissant 8GFlops FFT de performances s'avère fort intéressante. Nous notons aussi que nos deux architectures sont similaires au niveaux des capacités brutes de calculs, notamment pour 1FFT. Cependant, le GPU Haswell quant à lui, est doté de la mémoire *EDRAM* de 128MBytes qui lui permet d'avoir un meilleur rendement au niveau transferts de données. Ceci se traduit par une légère amélioration des performances quand on lance deux FFT en parallèle.

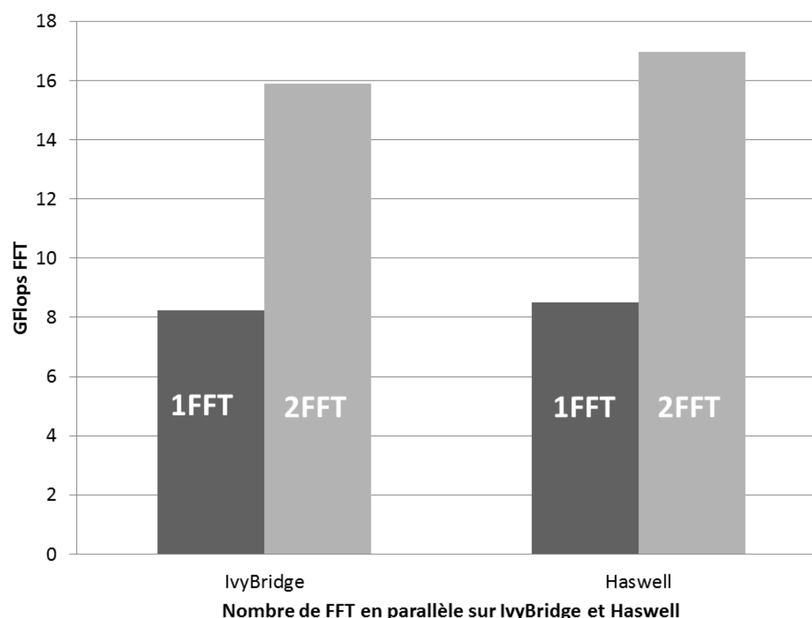


FIGURE 4.17 – Performances FFT 1024 sur GPU IvyBridge et Haswell

L'efficacité des calculs est calculée par rapport au maximum de performances que nous pouvons atteindre avec notre FFT sur notre architecture. Nous mesurons donc la performance des calculs seuls (sans inclure les transferts mémoire) sur une EU. Nous obtenons ainsi 10GFlops FFT, ce qui représente la performance maximum atteignable avec des transferts mémoires négligeables. La valeur ainsi obtenue sera

notre valeur de référence pour déterminer notre efficacité. Nous notons aussi que cette valeur passe très bien à l'échelle, pour calculer n FFT sur nEU indépendants, avec n entier (petit).

La figure 4.18 démontre néanmoins que l'efficacité se détériore légèrement avec le nombre d'EU utilisées pour IvyBridge (Nombre de FFT). Nous pouvons aussi constater que le GPU Haswell garde une quasi constance en terme d'efficacité avec l'augmentation des FFT en parallèle. Une efficacité de 80 % dénote alors l'aspect optimal de notre implémentation. Nous signalons aussi que jusqu'à présent, aucune implémentation n'atteint de telles performances sur cette architecture.

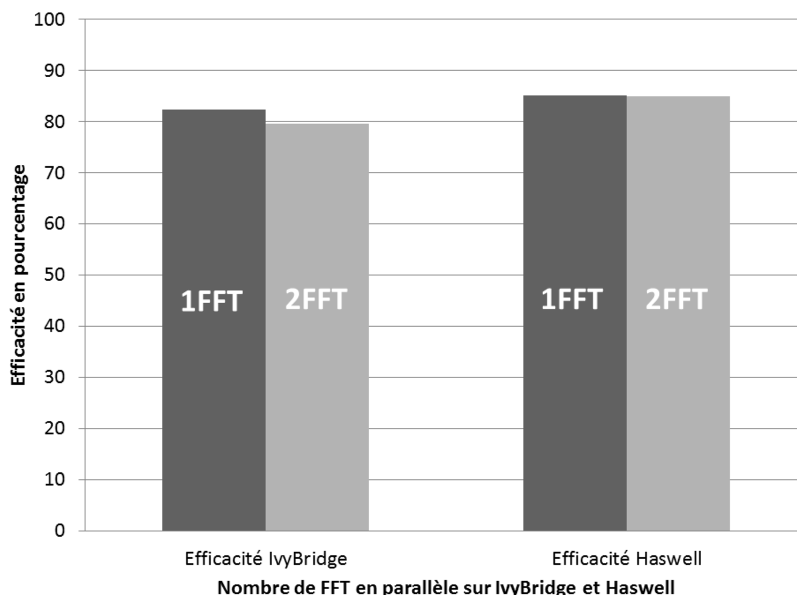


FIGURE 4.18 – Efficacité de notre implémentation FFT 1024 sur GPU IvyBridge et Haswell

Nous étudierons dans le chapitre 5 comment ces performances se mettent à l'échelle avec le nombre de EU, quand n atteint des valeurs comparables au nombre total d'EU disponibles.

4.2 Adaptation SIMD sur un cœur CPU

Nous étudions maintenant la forme SIMD, adaptée aux cœurs *Intels Core* CPU qui disposent de ce mécanisme, avec une largeur vectorielle de 256bits. À la différence du cas des GPU, ce type d'implémentation a déjà été largement étudié de manière académique et commerciale. Néanmoins, le travail d'adaptation de l'algorithme FFT sur les générations de CPU est un sujet encore d'actualité [33, 34, 35, 36] du fait de l'évolution constante des architectures CPU, notamment l'augmentation de la largeur des unités de calcul SIMD. Ces dernières viennent avec de nouveaux jeux d'instructions, qui nécessitent de réécrire toutes les applications (précédemment optimisées pour la génération antérieure) afin d'en tirer profit. Nous voulons en outre transposer notre approche de la section GPU précédente (regrouper autant d'étages que possible mais sans sortir des registres locaux d'un cœur unique) à ce cas CPU/SIMD.

Une thèse précédente à Kontron [9] avait comme sujet l'optimisation de la FFT sur des CPU de type PowerPC avec des unités SIMD de 128bits. Nous nous sommes basés au départ sur celle-ci pour exploiter les CPU de type Intel de dernière génération *Haswell*. Ces derniers sont munis d'unités SIMD AVX2 de largeur 256bits ainsi que d'instructions de multiplication addition fusionnée (FMA).

L'objectif principal dans ce cas est de maximiser l'occupation des unités SIMD en exploitant toute leur largeur tout au long des calculs sur un seul cœur CPU.

Contrairement au cas des GPU, sur les architectures CPU nous n'avons pas à notre disposition un espace registres conséquent. Notre principale limitation de taille FFT sera alors liée à l'utilisation du cache L1 qui est ici de taille 64KB. Nous nous limiterons aussi, pour des raisons de compatibilité avec l'implémentation GPU, aux FFT de taille 1024 points complexes simple précision.

Comme pour le cas des GPU, l'algorithme retenu se base sur celui de *Stockham*, et ce pour les mêmes raisons, en particulier éviter au maximum les ré-ordonnancements. Nous implémentons dans un premier temps un algorithme FFT en radix-2. Nous exploitons aussi la factorisation FMA étudiée section 2.1.3 pour nos cœurs de calcul dont nous allons maintenant détailler l'implémentation.

4.2.1 Cœur de calcul

Un cœur CPU Intel *Haswell* est doté de l'instruction vectorielle FMA ainsi que de l'instruction FNMA « multiplication négation puis addition » $fnma(a, b, c) = -(a \times b) + c$. Ces deux instructions ont la même latence [1] et la disponibilité de l'instruction FNMA nous permet de ne stocker qu'une seule version des *twiddle factors* alors que dans le cas des GPU nous étions obligés d'avoir deux versions, une avec le signe $+$ et une avec le signe $-$. Deux FMA ou FNMA peuvent s'éxécuter en parallèle sur deux vecteurs de 8 flottants chacun.

Le cœur de calcul s'écrira alors grâce aux *intrinsics* C d'Intel comme une succession de 6 FMA vectorielles permettant de traiter 8 papillons à la fois. Le code C correspondant est :

```
1  void coeurFFTVectoriel(__m256* srcRe1,__m256* srcRe2,__m256* srcIm1,
2                  __m256* srcIm2,__m256* dstRe1,__m256* dstRe2,
3                  __m256* dstIm1,__m256* dstIm2,__m256* wRe,__m256* wIm)
4  {
5          __m256 tmp1,tmp2;
6          tmp1=_mm256_fnmadd_ps(*wIm,*srcIm2,*srcRe2);
7          tmp2=_mm256_fmadd_ps(*wIm,*srcRe2,*srcIm2);
8          *dstRe1=_mm256_fmadd_ps(tmp1,*wRe,*srcRe1);
9          *dstRe2=_mm256_fnmadd_ps(tmp1,*wRe,*srcRe1);
10         *dstIm1=_mm256_fmadd_ps(tmp2,*wRe,*srcIm1);
11         *dstIm2=_mm256_fnmadd_ps(tmp2,*wRe,*srcIm1);
12 }
```

Ce cœur de calcul est la brique de base utilisée tout au long de ce chapitre. Cette dernière est optimale tant au niveau complexité de calcul (comme vu section 2.1.3) qu'au niveau occupation des unités SIMD (ici 100% d'utilisation).

Pour ce faire, nous avons fait le choix d'organiser les données en mémoire de manière contiguë et non entrelacée. Nous avons ainsi deux tableaux de données, un

1. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

pour la partie réelle et un pour la partie imaginaire. Ce choix a été fait afin d'exploiter de manière simple toute la largeur de l'unité SIMD. En effet, chaque opération vectorielle est effectuée entre deux vecteurs distincts, ce qui produit alors 8 papillons à l'aide de 6 instructions vectorielles. En revanche, dans l'approche entrelacée, chaque point complexe est stocké en entrelaçant les parties réelles avec les parties imaginaires, et on ne traite que 4 papillons. La figure 4.19 décrit la manière de calculer une multiplication complexe (un papillon) avec 6 opérations (3 multiplications et 3 additions ainsi qu'une permutation).

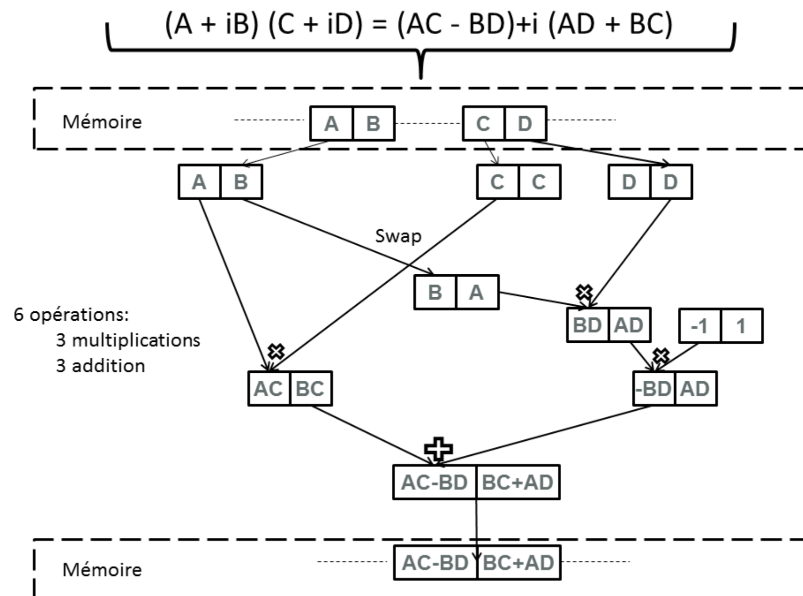


FIGURE 4.19 – Optimisation papillon FFT entrelacé : 6 opérations flottantes + une permutation

Dans cette thèse, nous adopterons donc l'approche FMA (non entrelaçée) pour les raisons citées précédemment. Deux tableaux de données nous serviront pour stocker la partie imaginaire et la partie réelle.

4.2.2 Première implémentation SIMD

Puisque nous avons choisi d'utiliser l'algorithme de *Stockham* avec une décimation en fréquence (DIF), nous obtenons un rapprochement progressif des indices de chaque papillon au fil des étages. Sachant aussi que nous pouvons calculer 8 points flottants simple précision en parallèle avec chaque unité SIMD *AVX2 (256bit)*, 8 papillons consécutifs peuvent donc être chargés directement comme illustré sur la figure 4.20.

Les premières passes de la FFT ne nécessitent alors aucune réorganisation des données. Un simple chargement aligné des données est effectué afin de recharger les registres vectoriels de l'unité SIMD.

Il faudra $N/16$ itérations pour calculer tous les points d'une FFT de taille N, chaque itération décale l'indice de départ de 8 positions (voir figure 4.21).

Cette procédure est ainsi répétée pour chaque étage de la FFT, à l'exception des

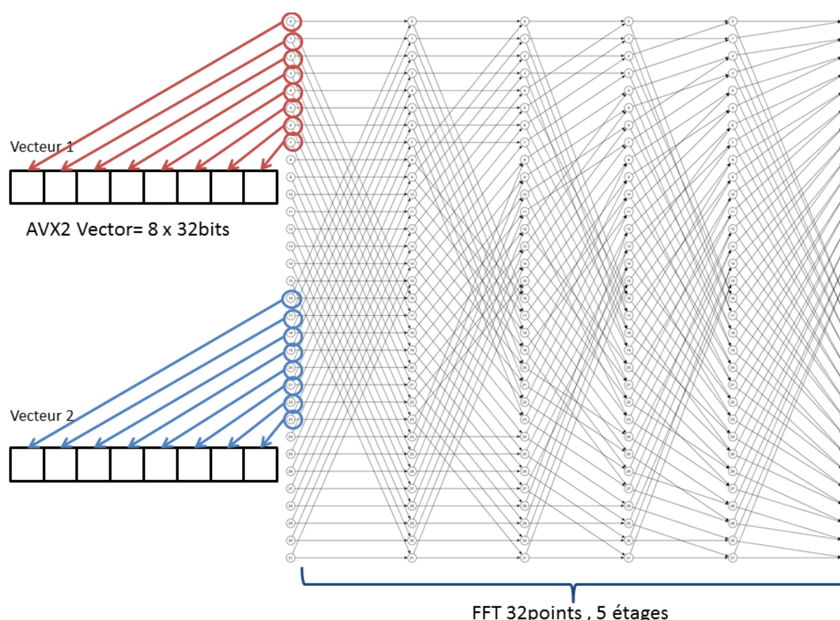


FIGURE 4.20 – Vectorization AVX2 de la FFT 32 points complexes

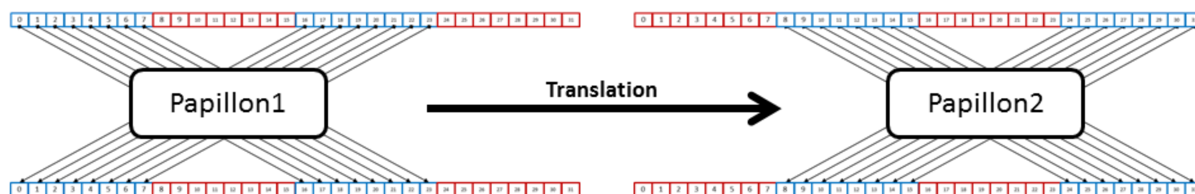


FIGURE 4.21 – Vectorisation AVX de la FFT et translation des calculs avant les quatre dernières passes

4 dernières passes où un ré-ordonnancement des données s'impose. Les données nécessaires pour calculer 8 papillons sont alors trop rapprochées (physiquement en mémoire) à partir des $\log_2(8 \times 2) = 4$ dernières passes (8 papillons FFT produisent 16 points). Donc une simple lecture de 8 points consécutifs n'est plus suffisante. Il faut donc réorganiser les données dans les vecteurs.

Nous utilisons alors les instructions de permutation fournies par le jeux d'instructions AVX2 afin de charger les bons points dans chaque vecteur. Le source suivant contient la structure de la dernière passe (étage 10) de la FFT.

```
1        //stage 10
2        index1=0;        index2=8;
3        index3=0;        index4=512;
4        mask1=136;       mask2=221;
5        for(i=0;i<64;++i)
6        {
7        pref1=512+(i*8);
8        //Load data from destination + permutations
9        tmpRe1=_mm256_loadu2_m128(dstRe+index2,dstRe+index1);
10       tmpRe2=_mm256_loadu2_m128(dstRe+index2+4,dstRe+index1+4);
11       tmpIm1=_mm256_loadu2_m128(dstIm+index2,dstIm+index1);
```

```
12         tmpIm2=_mm256_loadu2_m128(dstIm+index2+4,dstIm+index1+4);
13         srcRe1=_mm256_shuffle_ps(tmpRe1, tmpRe2, mask1);
14         srcRe2=_mm256_shuffle_ps(tmpRe1, tmpRe2, mask2);
15         srcIm1=_mm256_shuffle_ps(tmpIm1, tmpIm2, mask1);
16         srcIm2=_mm256_shuffle_ps(tmpIm1, tmpIm2, mask2);
17         //Load twiddle factors
18         twiddleRe=_mm256_load_ps(wRe+pref1);
19         twiddleIm=_mm256_load_ps(wIm+pref1);
20         //perform the FFT computation
21         coeurFFTVectoriel()
22         //store date to destination vector
23         _mm256_store_ps(srcRe+index3,dstRe1);
24         _mm256_store_ps(srcIm+index3,dstIm1);
25         _mm256_store_ps(srcRe+index4,dstRe2);
26         _mm256_store_ps(srcIm+index4,dstIm2);
27         //update the offset value
28         index2+=8;index3+=8;index4+=8;
29         //update indexes for the next loop
30         index1=index2;
31         index2+=8;
32         }
```

Nous remarquons qu'il est nécessaire d'utiliser trois instructions pour arriver à charger correctement un vecteur source (voir ligne 9-16). Nous notons aussi que l'écriture suit toujours le même schéma, aucune modification n'est nécessaire par rapport aux autres passes.

Les performances obtenues pour cette solution actuelle en comparaison avec la bibliothèque propriétaire d'Intel IPP sont équivalentes à 5% près (voir figure 4.22). Nous avons donc démontré par cette implémentation que nos choix d'algorithmes et d'optimisation sont à la hauteur de l'état de l'art des bibliothèques disponibles sur le marché, et nous somme confiant qu'avec quelques autres optimisations plus avancées nous pourrions obtenir des performances supérieurs à la bibliothèque IPP d'Intel.

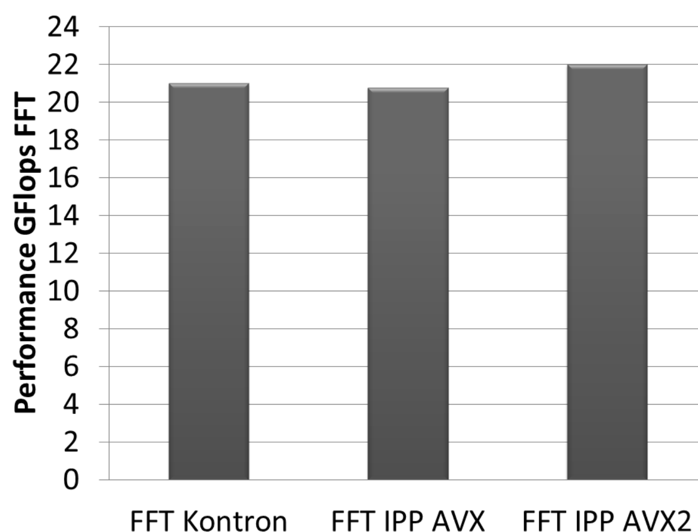


FIGURE 4.22 – Comparaison entre notre première optimisation et la version native IPP d'Intel (en utilisant l'AVX2)

Par ailleurs l'implémentation native *IPP* d'*Intel* semble ne pas tirer avantage des capacités architecturales AVX2 (cf section 3.2) par rapport à AVX original. Ceci nous indique de nouvelles pistes pour notre optimisation.

Nous avons ainsi pu constater que le gain n'est pas très significatif par rapport à ce que rapporte réellement la nouvelle unité SIMD AVX2. L'unité SIMD AVX n'est pas dotée d'instruction FMA tandis que l'AVX2 en est dotée. Nous avons soupçonné le fait que l'implémentation de la bibliothèque FFT d'Intel IPP n'est pas très optimisée pour exploiter l'instruction FMA.

Nous avons alors analysé les performances obtenues par notre implémentation en fonction de la passe pour voir d'où vient le ralentissement de notre implémentation.

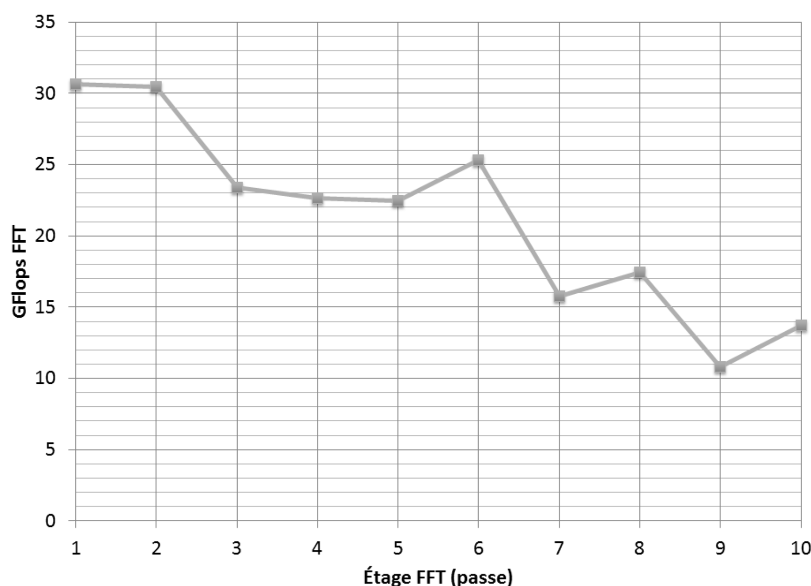


FIGURE 4.23 – Performances par passes FFT sur CPU

Notre analyse a permi de mettre en évidence le fait que les quatre dernières passes sont les plus pénalisantes pour notre implémentation comme le montre la figure 4.23. Ceci peut sembler étonnant à première vue, chaque étage calculant exactement le même nombre de « papillons » , mais vient du fait de la complexité grandissante du nombre de *twiddle factors* impliqués, ainsi que l'obligation d'utiliser des instructions de permutation.

Nous avons donc décidé de regrouper les quatre dernières passes en un seul algorithme, ce que nous autorise la taille des vecteurs SIMD AVX2 en gardant les valeurs dans les registres locaux (en nombre suffisant). Nous aurions aussi pu généraliser ce regroupement comme nous l'avons fait sur GPU pour la version (3,3,2,2) étages, par exemple en calculant (2,4,4) étages. Ce travail n'a pas été conduit faute de temps. Nous présentons ici la version (1,1,1,1,1,1,4) étages.

Dans cette version (1,1,1,1,1,1,4), la permutation effective des valeurs avant la quadruple phase finale revêt alors une importance extrême. Le ré-ordonnancement des données est effectué à l'aide d'instructions de permutations vectorielles qui existent dans le jeu d'instructions du processeur.

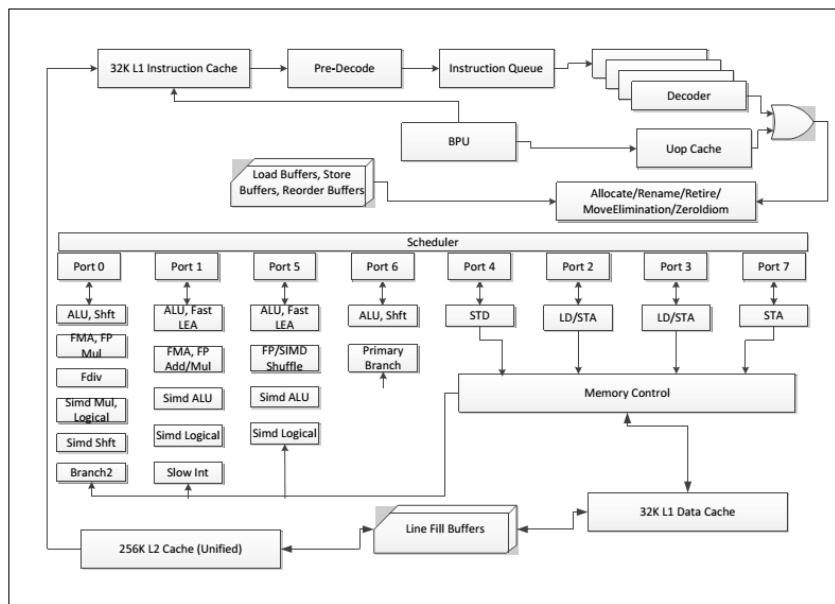


FIGURE 4.24 – Architecture d'un cœur CPU Haswell d'Intel

Le processeur Intel Haswell est doté effectivement de deux unités FMA sur les ports 0 et 1, comme le montre la figure 4.24, il est aussi doté d'instructions de permutation (*Shuffle*) sur le port 5. Nous pouvons vite constater qu'une mauvaise utilisation de l'instruction de permutation peut nuire grandement aux performances et rendre le port 5 un goulot d'étranglement.

Une des particularités des instructions de permutation actuelles SIMD x86 de l'architecture *Intel Core* est qu'elles portent sur des vecteurs de 8 valeurs, alors que nous avons besoin de permuter 16 valeurs (selon un schéma précis). Reste donc à déterminer quelle combinaison pratique de petites permutations autorise le plus efficacement à représenter notre grande permutation (sachant que ces petites permutations ont chacune un coût différent renseigné par Intel dans sa documentation technique).

Le listing suivant résume les instructions de permutation les plus utiles dans notre cas, ainsi que leurs coûts :

```
1  __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int imm8); //Latency: 1
2  __m256 _mm256_blend_ps(__m256 a, __m256 b, const int imm8); //Latency: 1
3  __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);//Latency: 2
4  __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int imm8); //Latency: 1
5  __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int imm8);//Latency: 3
6  __m256 _mm256_permutevar8x32_ps (__m256 a, __m256i idx);//Latency: 3
7  __m256 _mm256_permutevar_ps (__m256 a, __m256i b); //Latency: 1
8  __m256 _mm256_permute_ps (__m256 a, int imm8); //Latency: 1
9  __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);//Latency: 1
10 __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);//Latency: 1
11 __m256 _mm256_movehdup_ps (__m256 a); //Latency: 1
12 __m256 _mm256_moveldup_ps (__m256 a); //Latency: 1
```

Nous investiguons dans la partie suivante cette approche.

4.2.3 Deuxième implémentation SIMD : Adaptation des permutations

Mapper une permutation sur un jeu d'instructions est un problème d'adaptation d'algorithme, il illustre bien la nécessité de trouver le bon compromis pour atteindre les meilleures

performances possibles pour un hardware donné.

Donc optimiser les quatre dernières passes revêt une importance capitale dans notre implémentation. Nous optimiserons alors ces passes en les regroupant.

Les quatre dernières passes correspondent naturellement à une FFT de taille 16, Nous aurons alors besoin de deux registres 256bits pour effectuer tout le calcul.

La figure 4.25 illustre la structure de notre approche, les premières passes garderons alors la même structure que l'approche précédente. Chaque étage nécessitera alors 64 chargements de registres et 64 écritures dans la mémoire. Les 4 dernières passes nécessiterons un seul chargement de données et une seule écriture des données, ce qui augmente la localité des calculs.

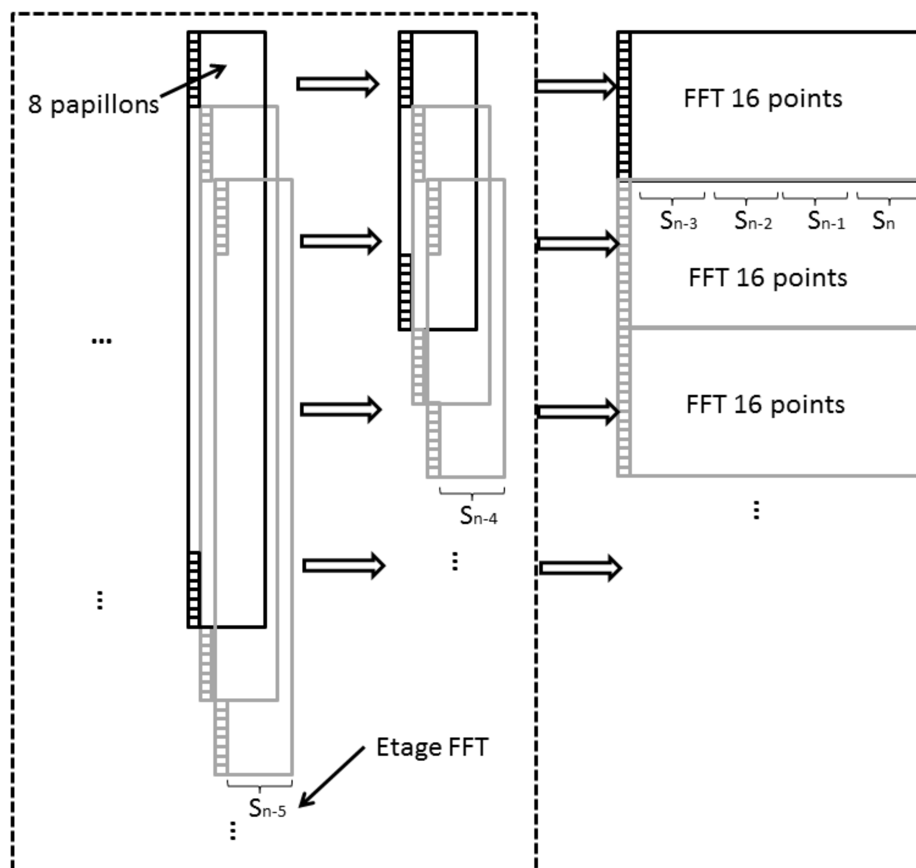


FIGURE 4.25 – Organisation des calculs d'une FFT sur un CPU à l'aide des unités SIMD

Afin d'arranger au mieux les calculs de notre bloc des 4 dernières passes, plusieurs solutions s'offrent à nous. L'essentiel est d'obtenir le bon résultat pour ce bloc. Nous sommes donc libres d'organiser les données dans chaque vecteur. Il faut juste s'assurer que les valeurs opposées dans les deux vecteurs font partie des valeurs d'un papillon.

Nous avons besoin de calculer une FFT de 16 points complexes, cette dernière nous définit aussi la transformation à effectuer sur les données tout au long des quatre dernières passes. Le graphe figure 4.26 schématise les dépendances de données : Les couleurs bleu et rouge correspondent au deux registres V1 et V2 de largeur 8 (points flottants), chaque registre doit contenir la moitié des valeurs pour un papillon.

Nous avons constaté que deux arrangement des données sont possibles. Le premier est l'application directe du graphe en figure 4.26. Les opérations de permutation sont différentes en fonction de la passe (voir figure 4.27).

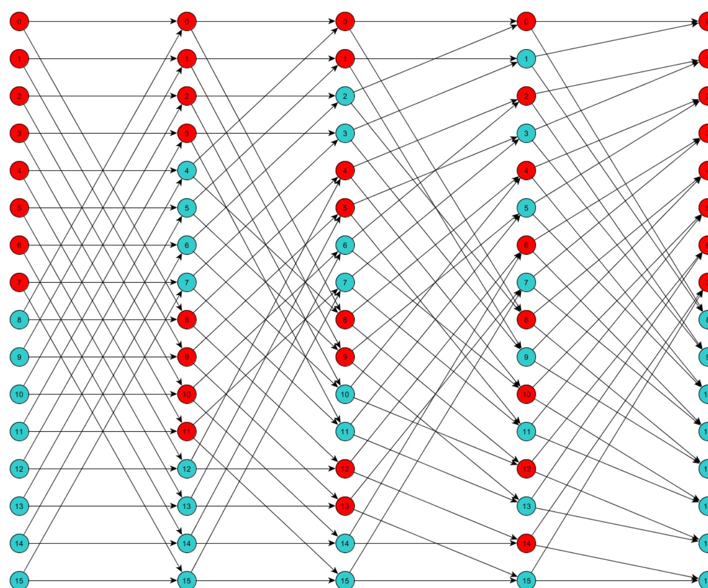


FIGURE 4.26 – Les transformations vectorielles pour effectuer les quatre dernières passes de la FFT : en Rouge le vecteur V1 et en bleu le vecteur V2

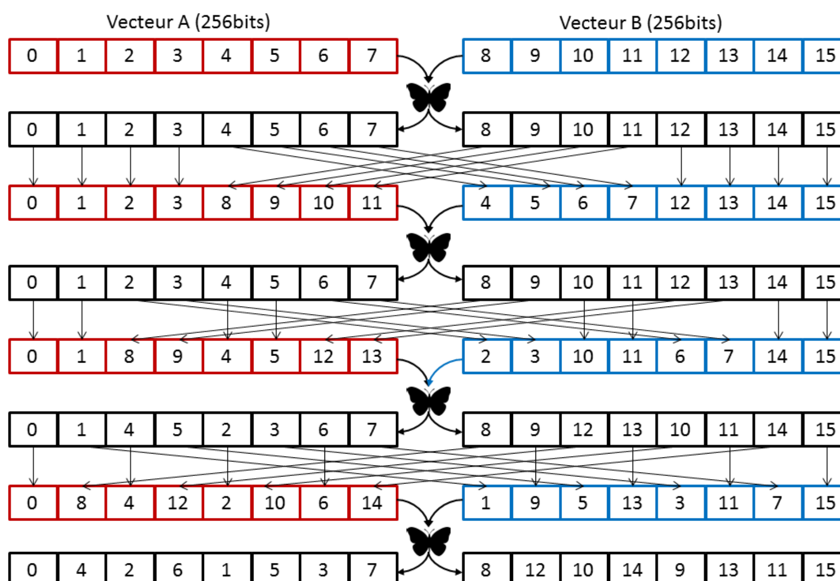


FIGURE 4.27 – Les transformations vectorielles pour effectuer les quatre dernières passes de la FFT : en rouge le vecteur A et en bleu le vecteur B

Nous savons aussi à travers la documentation technique sur le jeux d'instructions du CPU Intel Core Haswell que les instructions de permutation / manipulation des vecteurs ont des coûts différents (une latence de 1 à 3). Les instructions de permutation fournies par le jeux d'instructions du CPU sont accessibles par l'intermédiaire des fonctions *intrinsics* C. Nous présentons dans le listing suivant 12 d'entre elles utiles pour notre thèses. Ces instructions permettent de permuter des vecteurs de 256Bits.

```
        //Permutation constante
        tmpRe1=_mm256_unpacklo_ps(srcRe1, srcRe2);
        tmpRe2=_mm256_unpackhi_ps(srcRe1, srcRe2);
        mask1=2; // 00000010=2
        mask2=49; // 00110001=49
        srcRe1=_mm256_permute2f128_ps(tmpRe1, tmpRe2, mask1);
        srcRe2=_mm256_permute2f128_ps(tmpRe2, tmpRe1, mask2);
```

Notre deuxième alternative est alors un arrangement dit à géométrie constante, c'est-à-dire que toutes les permutations seront les mêmes en fonction de la passe. La figure 4.28 montre les permutation à effectuer pour calculer la FFT. Nous avons fait ce choix afin d'avoir un coût fixe pour toutes les passes, et aussi pour simplifier notre code source.

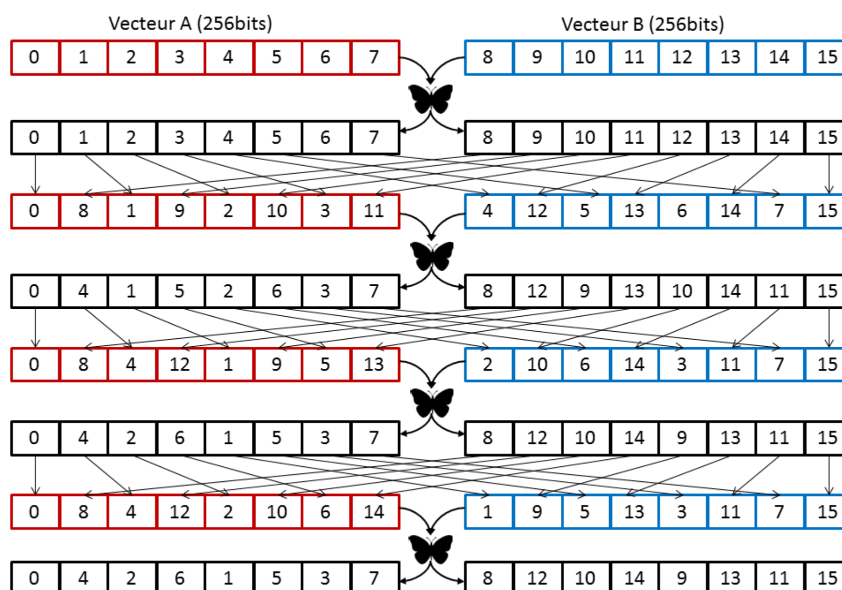


FIGURE 4.28 – Les transformations vectorielles pour effectuer les quatre dernières passes de la FFT : en rouge le vecteur A et en bleu le vecteur B, on utilise une transformation dite constant géométrie

4.2.4 Expérimentation et résultats pratiques

La figure 4.29 illustre les performances par passe obtenues en agissant sur les 4 dernières passes. Nous constatons une nette amélioration des performances de ces passes par rapport à l'implémentation initiale naïve. Nous notons aussi que les performances de ces quatre dernières passes sont mesurées de manière inégale entre elles. Nous ne tirerons aucune conclusion par rapport à chaque passe de ces quatre dernière, mais nous les traiterons comme un bloc compact.

La figure 4.30 aussi un gains en performances sur ces dernières passes assez minime, mais qui à le mérite de simplifier notre code source.

Nous obtenons alors les performances suivantes en figure 4.31. Notre FFT mixed-radix démontre que l'utilisation de l'optimisation FMA, que nous avons présentée au chapitre 2, ainsi que le regroupement des 4 dernières passes fournit un gain important de performances 10% par rapport à la meilleure bibliothèque sur le marché d'Intel (IPP). Nous signalons aussi que

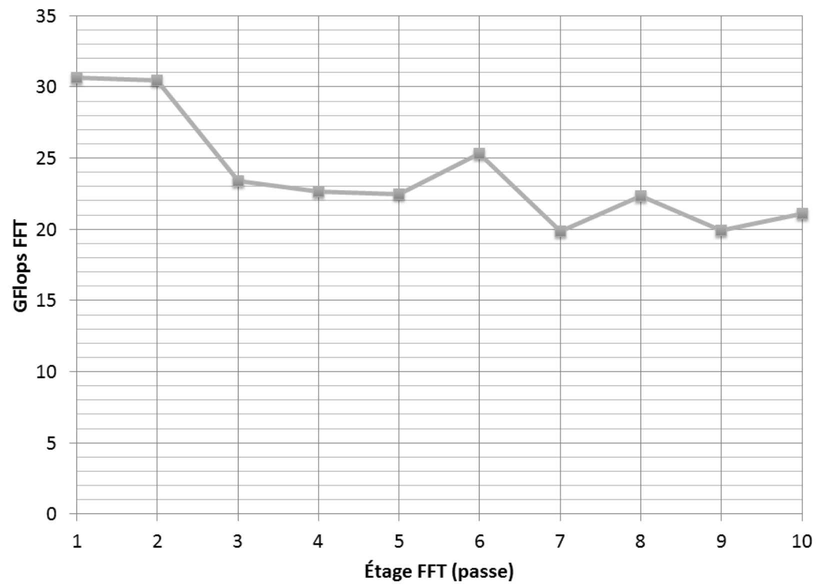


FIGURE 4.29 – Performances par passes FFT sur CPU après optimisation des quatre dernières passes

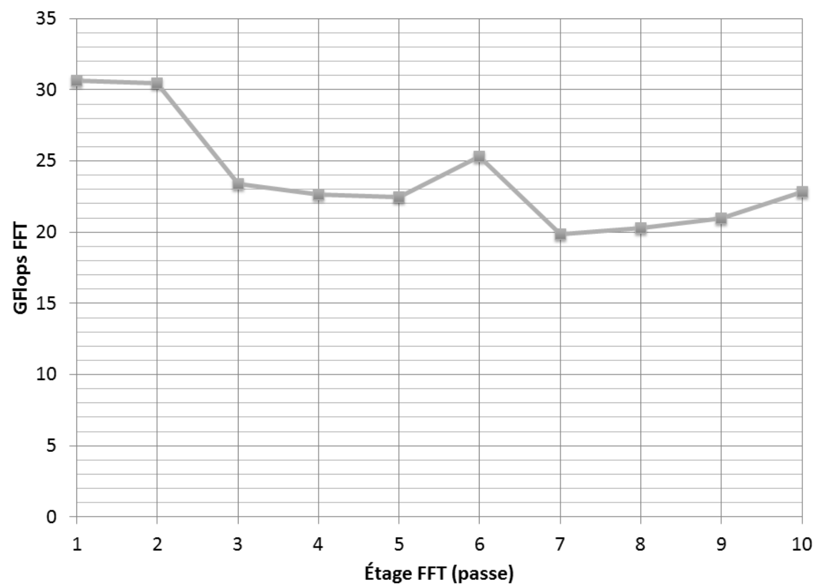


FIGURE 4.30 – Performances par passes FFT sur CPU après optimisation des quatre dernières passes

pour nos mesures nous désactivons le mode *Intel Turbo Boost*, et fonctionnant en fréquence nominale.

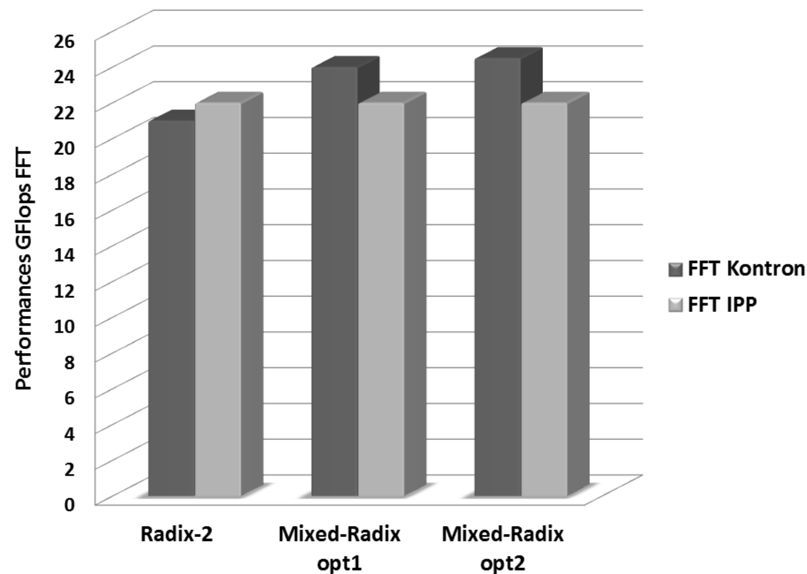


FIGURE 4.31 – Performances de la FFT sur CPU

Bilan et discussion

Ce chapitre résume notre travail principal, qui consiste à ajuster au mieux le calcul d'une FFT de taille donnée (ici 1024) à la taille et à la forme d'un cœur élémentaire (CPU ou GPU), afin de conserver autant que possible les valeurs locales dans les registres. Le but n'était pas de calculer au plus vite une FFT unique, mais d'avoir des modules compacts pour combiner efficacement ensuite de nombreuses FFT réparties sur tous les cœurs.

On aurait pu souhaiter écrire un code unique pour CPU et GPU, ce qui se révèle actuellement infaisable pour plusieurs raisons, largement du fait des lacunes du compilateur courant OpenCL trop naïf sur *Intel Core* : le code SIMD se retrouve scalarisé dans le compilateur GPU, et le code SIMT est très imparfaitement vectorisé. L'architecture CPU SIMD permet d'enchainer localement 4 étages de calcul, et l'architecture GPU SIMT (seulement) 3. Nous organisons l'expansion algorithmique et de gestion des décalages d'indices en fonction de ces paramètres.On peut noter que, même si actuellement nous avons fait ces travaux « à la main » , ils sont a priori automatisables, ne dépendant que de 2 paramètres simples (taille FFT, nombre d'étages compactables en fonction des registres).

Chapitre 5

Performances système et distribution de l’application globale

Motivations

Le chapitre précédent a montré comment optimiser le placement/ordonnancement du calcul d’une FFT individuelle sur un cœur unique de calcul (CPU ou GPU), afin de calculer au maximum avec des variables locales. Par ce biais, les travaux pouvaient être relativement déterministes et analytiques pour le dimensionnement du parallélisme.

Nous devons maintenant étudier comment placer les nombreuses FFT opérant sur un flot d’images en entrée, certaines en parallèle et d’autres en séquence/*streaming*, afin de réaliser l’application radar en utilisant un nombre restreint de ressources (comptées en nombre de processeurs *Intel Core*).

Contrairement au cas précédent, nous allons ici rencontrer des problèmes de mouvement de données à travers des (hiérarchies de) mémoires complexes, avec des phénomènes de contention de bus et d’embouteillages de données, et aussi des problèmes de dissipation de chaleur.

Il faudra également prévoir que certains des cœurs CPU doivent pouvoir rester disponibles pour exécuter d’autres tâches moins gourmandes en relation avec le contexte de l’application.

Dans une première partie nous étudions de manière expérimentale, par la mesure, l’importance d’un certain nombre de phénomènes, qui ne sont pas forcément particuliers à l’application radar, mais interviendrons dans son déploiement.

Dans la seconde partie, nous nous focalisons plus sur cette application, sur le nombre d’opérations nécessaires dans un contexte temps-réel, et définissons une répartition efficace des calculs entre CPU et GPU. On ne peut a priori pas ici parler d’optimalité, au sens où il n’y a pas de critère fort et que certaines limites choisies pour éviter les phénomènes décrits dans la première partie peuvent être excessivement conservatives. Mais vu la granularité de la question générale (combien de processeurs *Intel Core* sur la carte), on peut juger (au sein de Kontron) si la variabilité de la solution trouvée reste acceptable.

5.1 Charges utiles de calculs FFT

Une approche naïve de la problématique d’allocation / mapping pourrait suggérer que, une fois établie la forme précise du code utilisé pour le calcul de la FFT 1024, tant sur (mono-

cœur) CPU que sur (*single* EU) GPU, on puisse mesurer expérimentalement sa performance, puis extrapoler qu'on peut exécuter autant de FFT que de cœurs disponibles (avec des performances identiques, tant sur CPU que sur GPU), pour résoudre la contrainte linéaire qui établit combien de processeurs *Intel Core* seront nécessaires pour effectuer l'ensemble des calculs (en établissant le bon ratio entre CPU et GPU, suivant des critères auxiliaires de puissance consommée ou autre).

La situation réelle est bien plus complexe, car des phénomènes limitatifs impérieux interviennent au niveau du système. Nous allons voir par exemple que la hausse excessive de température due à la dissipation thermique provenant des calculs limite (dans le cas précis de l'exécution de notre implantation de FFT) à la moitié du nombre de cœurs (CPU principalement) qui peuvent *safely* opérer en parallèle, alors que le plafonnement du débit de transfert de données entre mémoire partagée et GPU limite également à la moitié le nombre d'EU GPU qui peuvent opérer sans être sensiblement ralenties par l'attente de leurs données en entrée. L'objet de la présente section est de décrire comment ces phénomènes ont été étudiés expérimentalement.

5.1.1 Sur multicœur CPU

Notre approche d'optimisation de la FFT sur un cœur CPU étudiée au chapitre 4, a permis d'obtenir une implantation efficace, exploitant toute la puissance de calcul fournie par l'architecture CPU *Intel Core*. La nature indépendante de cette implantation permet d'avoir une liberté de placement et de *scheduling* sur les quatre cœurs disponibles sur notre CPU. Nous pouvons alors enlacer 4 FFT 1024 points complexes en parallèle sur chacun des cœurs.

5.1.1.1 Performances CPU

La performance FFT obtenue sur notre cœur CPU *Intel Haswell* à 2GHz pour notre implantation est de 24GFlops FFT. Cependant, cette performance est obtenue en exerçant le cœur CPU sur les mêmes données. Dans les cas pratiques, nous calculons une FFT sur des données différentes et les pénalités induites par les caches *miss* réduisent les performances. Nous obtenons alors dans notre cas une performance de 17GFlops FFT (cf. section 2.2.7). Nous pouvons alors calculer une FFT en 3011 ns (nanosecondes).

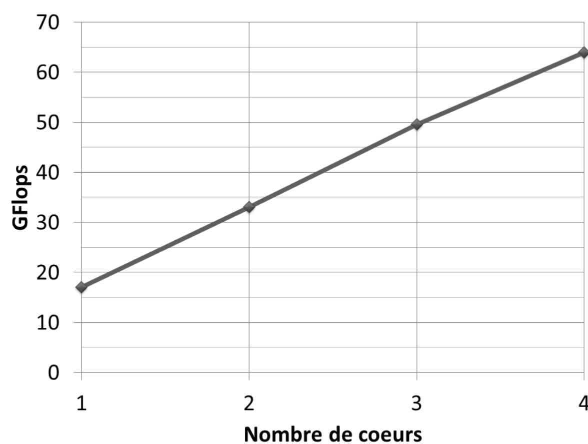


FIGURE 5.1 – Passage à l'échelle de notre FFT 1024 points sur CPU *Intel Haswell* (*Turbo* OFF et *Hyperthreading* OFF)

Nous vérifions aussi comment les performances se mettent à l'échelle si on augmente le nombre de cœurs utilisés (au maximum 4). Nous lançons alors sur chaque cœur une FFT indépendante sur des données différentes. La figure 5.1 illustre les résultats obtenus.

Nous pouvons alors calculer 4 FFT de taille 1024 points au débit de 64GFlops FFT. Ce qui veut dire 4 FFT en parallèle en 3200 ns sur notre CPU.

5.1.1.2 Dissipation thermique et consommation

Comme la FFT est un algorithme de calcul intensif, charger au maximum les cœurs CPU et GPU des processeurs va causer une hausse importante de température, puis de la consommation, jusqu'à la rupture possible (arrêt du processeur avant surchauffe, ou dégradation des performances "*throttling*"). Nous avons mesuré ces phénomènes tant pour GPU que pour CPU.

Ces tests expérimentaux, facilités par l'outillage technique disponible au sein des laboratoires Hardware de Kontron, nous ont permis de mettre en lumière d'autres effets influant sur les performances, à savoir la consommation électrique.

À l'aide d'enceintes climatiques, nous avons mis au point une expérimentation qui permet de mesurer l'impact de la température sur la consommation électrique. Nous avons fait varier graduellement la température sur un GPU embarqué AMD E6760 (finesse de gravure 40nm) qui exécute notre test de GFlops. Nous avons mesuré à des intervalles réguliers la consommation électrique de notre GPU. La figure 5.2 nous montre qu'un effet d'emballement de la consommation électrique du circuit graphique apparait avec l'augmentation de la température. On note que la consommation électrique augmente exponentiellement dès que le circuit dépasse un seuil de température donné, on note aussi que 10 watt sont consommés juste en dépassant les 95°C.

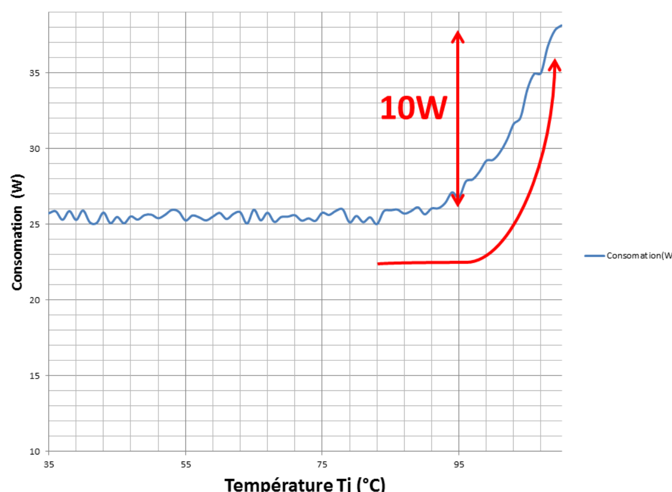
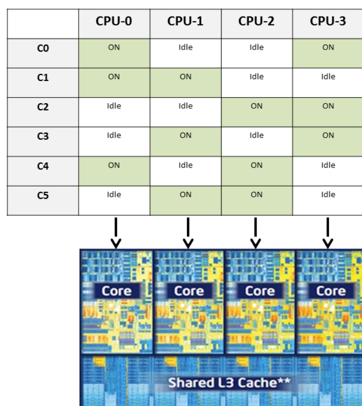


FIGURE 5.2 – Effet d'emballement thermique à 95°C qui fait augmenter la consommation électrique de 10 W

Nous notons aussi que ces observations sont aussi valables de manière plus prononcée dans le cas des CPU, qui ont une finesse de gravure plus petite (22nm dans notre cas *Intel Haswell*).

Nous avons aussi remarqué qu'on peut réduire la température du SoC en répartissant les



	CPU-0	CPU-1	CPU-2	CPU-3
C0	ON	Idle	Idle	ON
C1	ON	ON	Idle	Idle
C2	Idle	Idle	ON	ON
C3	Idle	ON	Idle	ON
C4	ON	Idle	ON	Idle
C5	Idle	ON	ON	Idle

FIGURE 5.3 – Expérience de placement des charges de calcul selon la localité du cœurs

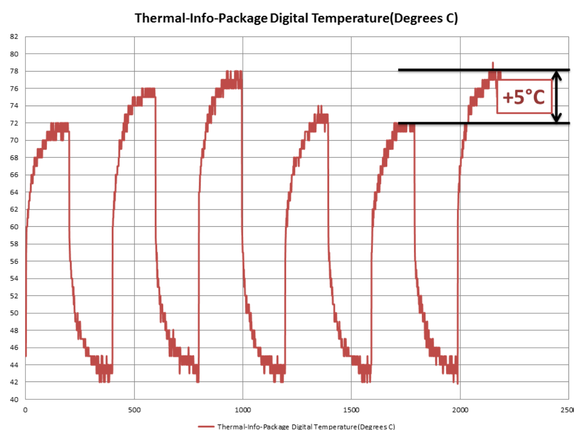


FIGURE 5.4 – Résultats thermiques sur le CPU IvyBridge d'Intel : 5°C de gain entre les cas favorables (C0,C3 et C4) et les cas défavorables (C1,C2 et C5)

calculs pour préserver des parties inactives du SoC, qui autoriseront alors une certaine dissipation thermique. Pour mettre en évidence cette hypothèse, nous avons imaginé l'expérience suivante sur notre processeur à quatre cœurs, comme indiqué sur la figure 5.3. Nous avons testé plusieurs répartitions spatiales de la charge de calcul (FFT 1024 points complexes) sur différents cœurs. Nous avons ainsi constaté un gain de 5°C quand les calculs sont effectués sur des cœurs espacés (cas c0, c3, c4) plutôt que dans des cœurs voisins (cas c1, c2 et c5)(voir figure 5.4.

Nous pouvons aussi ajouter à ces deux contraintes que le CPU peut être partiellement occupé par d'autres tâches plus légères mais essentielles ; la gestion de la pile de communication TCP / IP par exemple. Ainsi, il ne doit pas être saturé par du calcul de traitement de signal.

Nous concluons alors que notre CPU pourrait n'utiliser que la moitié des cœurs de calculs disponibles pour traiter les calculs FFT. Nous pouvons donc calculer 2 FFT 1024 points complexes en parallèle en 3100ns.

5.1.2 Sur GPU

Le GPU *Intel Core IvyBridge* contient 16EU, nous lançons sur chaque EU une FFT de 1024 points sur des données différentes. La courbe de performance figure 5.5 met en évidence une quasi-linéarité des performances jusqu'à la moitié du nombre d'EU, et une stagnation des performances au-delà. Nous obtenons alors 30GFlops FFT comme performance maximale sur le GPU *IvyBridge*. Nous pouvons donc calculer 16 FFT 1024 points complexes en parallèle en 27306 ns.

De manière équivalente, nous menons la même expérimentation sur le GPU *Intel Haswell* qui est doté cette fois ci de 40EU. Le même résultat est constaté sur cette architecture (voir figure 5.6). Nous obtenons alors 55GFlops FFT comme performance maximale sur le GPU *Haswell*. Donc nous pouvons calculer 40 FFT 1024 points complexes en parallèle en 51200 ns.

Nous remarquons aussi que les performances plafonnent à 55GFlops FFT à partir de la moitié des EU utilisées.

Afin de comprendre la cause de ce plafonnement des performances, sachant que les FFT

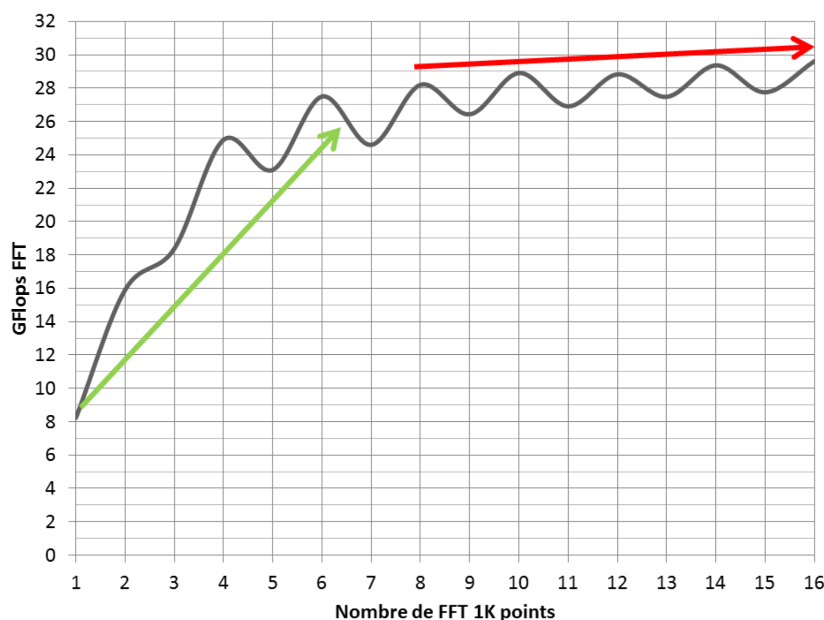


FIGURE 5.5 – Courbe des performances FFT 1024 points sur le GPU *Intel IvyBridge* à 16 EU

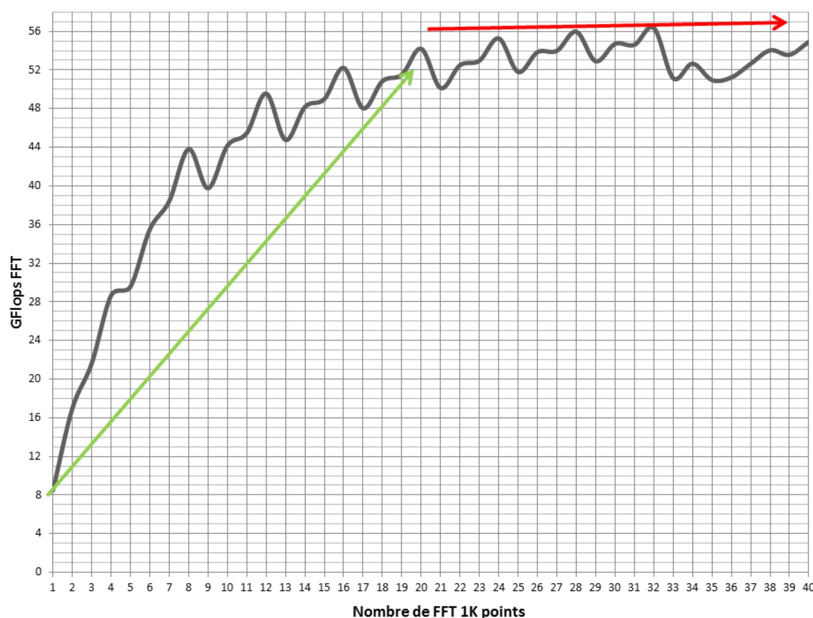


FIGURE 5.6 – Courbe des performances FFT 1024 points sur le GPU *Intel Haswell* à 40 EU

sont indépendantes, nous menons une analyse de notre implémentation FFT OpenCL. Nous avons caractérisé chaque partie de notre implémentation FFT. La figure 5.7 illustre la structure de notre algorithme. Nous avons isolé chaque partie et mesuré les performances obtenues pour chacune de ces phases.

Notre analyse à démontré que les transferts mémoires DRAM sont la cause principale de ce plafonnement. Effectivement, en mesurant les débits mémoires DRAM de ces deux architectures nous obtenons les résultats suivants :

— *Intel IvyBridge* : 5GB/s

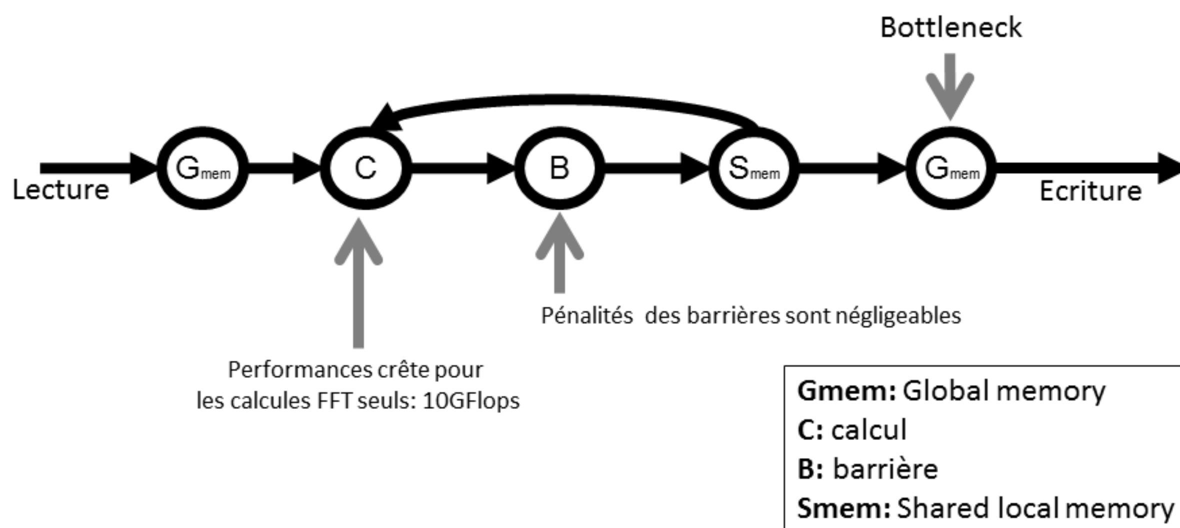


FIGURE 5.7 – Analyse de la structure de notre kernel FFT, et l'impact sur les performances de chaque phase

— *Intel Haswell* : 10GB/s

Sachant que pour un débit donné B, et une taille FFT donnée N nous pouvons calculer à l'aide de la formule 5.1 la performance maximale théorique atteignable T_{max} en GFlops FFT.

$$T_{max} = \frac{5 \cdot N \cdot \log_2(N) \times B}{2 \times 4 \times N} \tag{5.1}$$

Si on injecte les valeurs des débits mémoire de nos deux architecture dans cette formule nous obtenons les performances T_{max} respectives 32GFlops FFT pour *IvyBridge* et 62GFlops FFT pour *Haswell*. Ce qui veut dire que les performances FFT mesurées sur nos architectures plafonnent bien à cause du débit DRAM.

Le tableau 5.8 synthétise la correspondance entre le débit de données et les performances maximales théoriques T_{max} obtenues à l'aide de la formule 5.1. Nous pouvons dire aussi que les transferts mémoires sont le point critique de tout système parallèle.

		Gflops						
		20	**25**	**30**	**40**	**50**	**55**	
Taille de la FFT	**512**	3,6	4,4	5,3	7,1	8,9	9,8	**Gbyte/s**
	1024	3,2	4,0	5,0	6,0	8,0	8,8	
	2048	2,9	3,6	4,4	5,8	7,3	8,0	
	4096	2,7	3,3	4,0	5,3	6,7	7,3	
	8192	2,5	3,1	3,7	4,9	6,2	6,8	
	16384	2,3	2,9	3,4	4,6	5,7	6,3	
	32768	2,1	2,7	3,2	4,3	5,3	5,9	

PCI-e x 4	Gen 2	2 Gbyte/s
	Gen 3	4 Gbyte/s
PCI-e x 8	Gen 2	4 Gbyte/s
	Gen 3	8 Gbyte/s
PCI-e x 16	Gen 2	8 Gbyte/s
	Gen 3	16 Gbyte/s
Exceed PCI-express capability		>16 Gbytes/s

FIGURE 5.8 – Tableau de correspondance débit mémoire et performances FFT

Nous concluons par dire qu'afin d'éviter la congestion du bus de données, nous n'utiliserons que la moitié des EU GPU. Nous pouvons alors calculer 8 FFT 1024 points complexes en 13653 ns sur le GPU *IvyBridge* ; et 20 FFT 1024 points complexes en 20480 ns sur le GPU *Haswell*.

5.1.3 Conséquences

Pour des raisons différentes (expliquées précédemment) pour le CPU et le GPU, nous n'utiliserons que la moitié des cœurs CPU et la moitié des EU GPU (*Haswell*) pour calculer plusieurs FFT en parallèle. Notre objectif étant d'exécuter un traitement radar qui nécessite de calculer plusieurs FFT sur des flots de données. Nous sommes aussi confrontés à la problématique suivante : comment doit-on distribuer ces calculs sur nos cœurs CPU et GPU en respectant les exigences de l'application et les contraintes de l'architecture. Ceci est étudié dans la section suivante.

5.2 Dimensionnement pour application radar

Dans la section précédente nous avons établi que des hypothèse réalistes conduisent à estimer que la partie CPU multicœurs globale et la partie GPU (multi-EU) pouvaient effecteur respectivement 2 et 20 calculs de FFT dans 3100 et 20480 nanosecondes respectivement. Il nous reste donc à établir, connaissant les exigences temps-réel de l'application radar, le nombre raisonnable de processeurs *Intel Core* nécessaires pour cette fonction, et accessoirement la répartition effective des calculs. Il s'agit donc ici d'un problème d'optimisation relativement flexible, qui consiste pour partie à mesurer combien les ressources resteront disponibles pour d'autres tâches auxiliaires (ou pour de la redondance). La solution finalement adoptée consiste (on verra comment ci-dessous) à intégrer 3 processeurs *Intel Core* pour faire les calculs, en tenant la charge requise.

5.2.1 Besoins en calculs FFTs

Après la description de ces quelques expérimentations « in vivo » sur le circuit, afin de définir quelques paramètres pour calibrer notre implantation et sa répartition, nous en venons enfin au mode de placement et d'allocation des FFT élémentaires sur des cœurs de calcul pour paralléliser et optimiser l'application radar globale.

Nous décrivons maintenant le partitionnement de l'application radar (section 2.3 figure 2.16). Les images obtenues avec les traitements radar SAR sont des images carrées (256x256, 512x512, 1024x1024 ...). Ces différentes résolutions sont intimement liées à l'application (le profil de mission). Dans notre cas, notre application reçoit un flot d'images (1024x1024) au rythme de 1555 fois par seconde, et se doit de calculer 1024 FFT complexes 1D, elles-même de taille 1024. Le même traitement est itéré sur le flot de données continu provenant du module d'acquisition.

Tout l'algorithme est à base de traitements FFT, à l'exception du *corner turn* (transposition des données à $90°$). Cette étape contraint alors notre application à adopter un schéma parallèle dans chaque bloc de traitement (*Range* et *Azimut*), et un schéma séquentiel entre ces deux blocs. Cette étape sera exécutée exclusivement par le CPU qui orchestre toute l'application, elle dure 900ns pour transposer un tableau de 1024 x 1024 points complexes. Nous adoptons pour notre cas une transposition par blocs [37] en utilisant les quatre cœurs CPU.

Nous avons aussi des contraintes liées au temps réel, tout le traitement doit être effectué 1555 fois par seconde. Nous aurons alors 25GB de données à traiter par seconde par notre système.

La question est donc de répartir ces nombreux calculs de FFT (certains parallèles, d'autres non, suivant la fréquence de capture des données en entrée) (voir figure 5.9), et ce en prenant en compte et en respectant les phénomènes décrits en section 5.1.

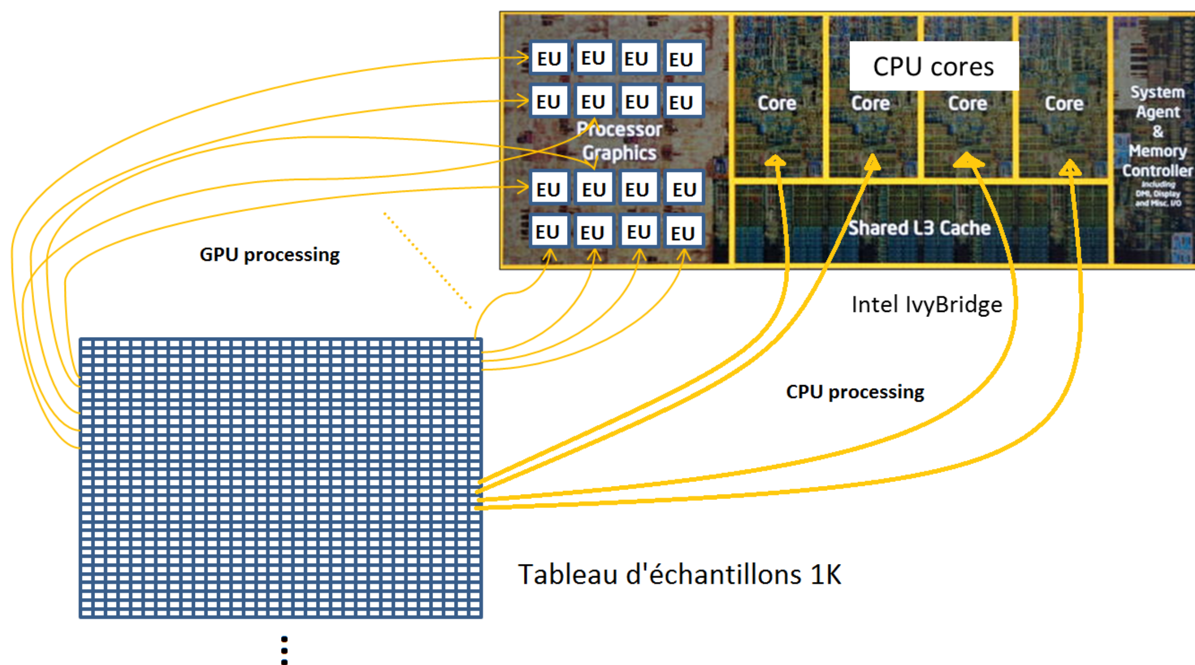


FIGURE 5.9 – Distribution des calculs FFT sur CPU et GPU (sur *Intel IvyBridge*)

5.2.2 Adaptation de la charge de calcul aux contraintes

Afin d'estimer la charge de calculs nécessaire à notre application, nous nous baserons sur les performances obtenues sur le CPU et GPU intégré *Intel Haswell* en 5.1.

Nous notons c alors le nombre de FFT s'exécutant sur le CPU en t_{cpu} nanosecondes, et g le nombre de FFT s'exécutant sur GPU en t_{gpu} nanosecondes. La charge de calcul totale quant à elle est notée q, soit le nombre de FFT que nous devons exécuter pour notre application. Nous notons aussi q_{cpu} le nombre de FFT que nous exécutons sur CPU et q_{gpu} le nombre de FFT que nous exécutons sur GPU.

Avec :

$$q_{cpu} + q_{gpu} = q \tag{5.2}$$

La contrainte de temps à respecter est notée r en nanosecondes. Nous noterons aussi en constante k les autres tâches de calcul ou de mouvements de données (ici le corner turn sur CPU). Donc pour respecter les contraintes de notre application nous les deux inégalité suivantes :

$$\begin{aligned} \frac{q_{cpu}}{c} \times t_{cpu} + k \leq r \\ \frac{q_{gpu}}{g} \times t_{gpu} \leq r \end{aligned} \tag{5.3}$$

Afin de trouver les valeurs de q_{cpu} et de q_{gpu} qui permettent de minimiser le temps de calcul r nous devons résoudre le système de contraintes suivant :

$$\begin{cases} q_{cpu} + q_{gpu} = q \\ \frac{q_{cpu}}{c} \times t_{cpu} + k \leq r \\ \frac{q_{gpu}}{g} \times t_{gpu} \leq r \\ q_{cpu} \geq 0 \\ q_{gpu} \geq 0 \end{cases} \tag{5.4}$$

Nous injectons nos valeurs dans le système 5.4, nous obtenons alors :

Sur CPU, nous pouvons calculer 2 FFT en 3100ns et nous pouvons aussi faire un *corner turn* en 900ns. Sur GPU, nous pouvons calculer 20 FFT en 20480ns.

— Si nous exécutons tous nos calculs sur CPU, nous aurons : $512 \times 3100 + 900 + 512 \times 3100 = 3175300ns \times 1555 = 5s$

— Si nous exécutons tous nos calculs sur GPU, nous aurons alors 3,2s

— Si nous exécutons nos calculs sur CPU et GPU en même temps, nous obtenons un temps de calcul inférieur à 2s pour le cas optimum ($q_{cpu} = 820$ et $q_{gpu} = 1228$). En moyenne nous obtenons 2,5s avec une charge de calcul équilibrée sur les deux (CPU et GPU). Nous prendrons ce cas pour avoir une estimation réaliste des performances.

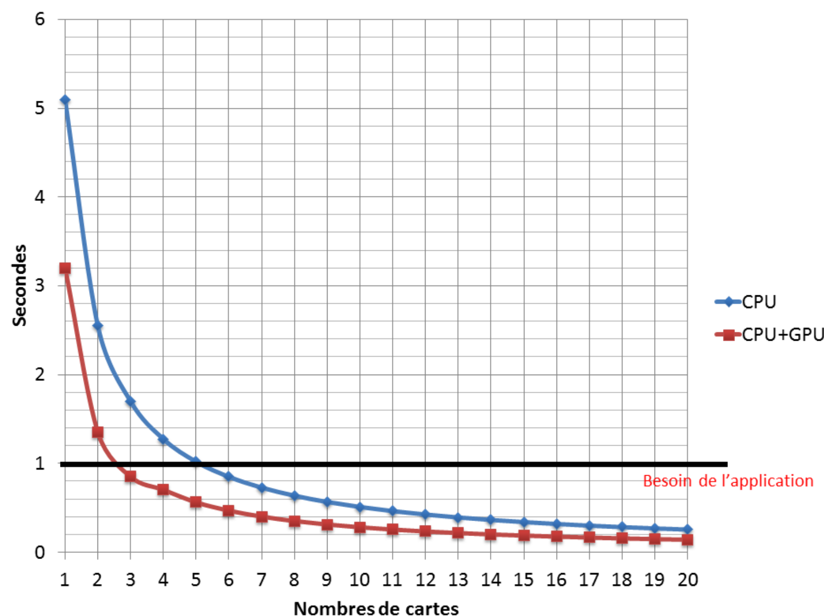


FIGURE 5.10 – Besoin de l'application en puissance de calcul afin d'avoir du temps réel (1555 fois /seconde)

La figure 5.10, montre que pour atteindre la contrainte du temps réel (1555 images par secondes) il est nécessaire d'utiliser 6 cartes CPU *Intel Haswell* (voir figure 5.11). Cependant, si on exploite les performances du GPU intégré en calculant également sur ses cœurs, nous pouvons alors réduire le nombre de cartes par deux.

L'utilisation du GPU intégré permet de réduire le nombre de cartes à utiliser pour atteindre les objectifs en performance de l'application (ici 1555 fois par secondes). Nous pas-

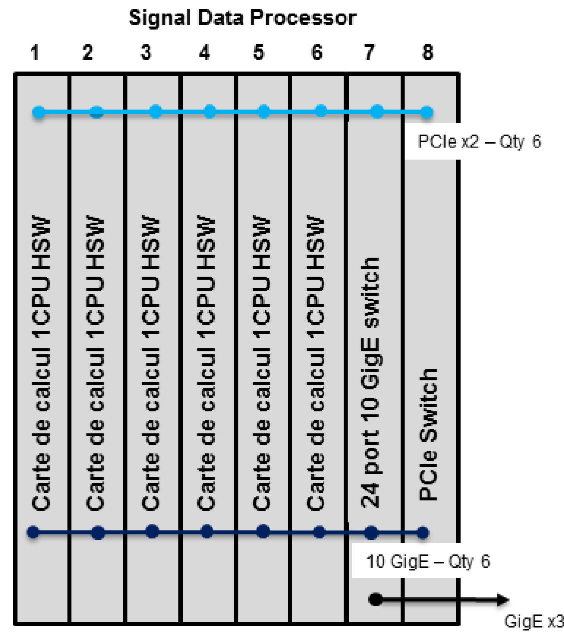


FIGURE 5.11 – Architecture de système de calcul radar à base de cartes CPU *Intel Haswell*

sons alors d'un système à 6 cartes (voir figure 5.11) à un système à 3 cartes. Le gain est alors double, les performances sont atteintes avec une consommation électrique et un poids moindre, le prix de revient du système est au passage réduit.

Bilan et discussion

Pour des raisons de temps concernant cette thèse les résultats de cette section restent préliminaires. Néanmoins notre résultat majeur sera que, en observant l'efficacité mesurée expérimentalement des temps de calcul, on constate une progression quasi-linéaire jusqu'à la moitié du remplissage des cœurs (GPU), puis une stagnation claire. Ce résultat semble indiquer qu'au-delà de cette densité de calcul, les communications de données ne peuvent plus alimenter les cœurs de calcul GPU. L'optimisation des ressources (et des équilibres et échanges entre GPU et CPU) peut se faire sous cette contrainte sans quasiment pénaliser le système.

Chapitre 6

Conclusion et Travaux Futurs

6.1 Related Works

Une requête sur un moteur de recherche avec les mots-clés "CPU GPU FFT" renvoie des centaines de milliers de liens.

Les articles les plus pertinents pour notre contexte semblent être [38, 39, 40, 41, 42].

Les sites communautaires fftw.org [1] et gpgpu.org [2] s'occupent respectivement de regrouper et recenser des efforts vers l'optimisation du calcul de la FFT et des calcul généralistes sur GPU, mais pas spécifiquement sur l'articulation des deux. Le projet Spiral [3] propose des implantations de fft très efficaces sur des architectures et accélérateurs matériels divers, mais certaines parties en sont brevetées et le code source non disponible. La situation est similaire pour la bibliothèque dédié Nvidia en Cuda CuFFT [4]. Pour la version CPU on peut mentionner pour notre contexte (en dehors de fftw et Spiral) la Math Kernel Library de Intel (Intel MKL) [5], qui propose des versions parallèles sur les architectures multi-cœurs.

La plupart du temps inspirées de la version *Stockham* de l'algorithme (avec variantes dans la parallélisation), la plupart des proposition précédentes cherchent à optimiser le calcul de la FFT au moyen de toutes les ressources disponibles. Elles optimisent parfois ces aspects en pratiquant une recherche dans un espace de solutions (restreint mais pluriel), par des techniques d'*auto-tuning* en particulier. Par contraste, notre approche cherche à limiter l'empreinte du design d'une FFT individuelle à la taille d'un cœur CPU ou d'un EU GPU, afin d'en faire une nouvelle brique de base dans un processus de compilation de haut niveau. Cette phase demande une étude spécifique et un ajustement manuel fin (avec aussi éventuellement des cas limités d'*auto-tuning* comme dans la décision de savoir comment répartir les 10 étages de la FFT 1024 pour l'implantation GPU en 2 blocs de 3 étages et 2 blocs de 2 étages. Il est important alors de s'assurer que le code sera effectivement exécuté de manière atomique (non-interruptible) en utilisant un cœur CPU ou GPU unique et entier, ce qui est parfois délicat en ne disposant que des instructions autorisées dans tel ou tel langage. Ce travail se révèle donc très délicat, pour assurer la localité des traitements et des données, source d'efficacité ; il a donc occupé l'essentiel du temps de cette thèse. Mais une fois obtenus, ces résultats permettent de définir une approche plus générale, à un niveau supérieur, où de vraies applications peuvent être placées et ordonnancées de manière efficace ("optimisée"), en s'appuyant sur des bibliothèques paramétriques pré-implantées (de FFT et pos-

1. `http://www.fftw.org`
2. `http://www.gpgpu.org`
3. `http://http://www.spiral.net`
4. `http://docs.nvidia.com/cuda/cufft/`
5. `https://software.intel.com/en-us/intel-mkl`

siblement dans le futur d'autres fonctions d'intérêt général, en algèbre linéaire etc). Il existe peu de travaux dans lesquels l'optimisation "locale" de bibliothèques spécialisées sur des architectures parallèles avec accélérateurs matériels est considérée à la fois pour son propre mérite mais également dans le contexte de programmes plus larges.

6.2 Développements futurs et perspectives

Faute de temps nous n'avons pas pu généraliser ou étendre les travaux du chapitre 5 pour prendre en compte des descriptions génériques d'applications, souvent présentées à la manière de graphes de tâches. L'approche AAA, préconisée au sein de l'équipe d'accueil Inria Aoste [6], utilise des descriptions de nature Réseaux de Processus flot de données pour cet objectif. Au sein du projet Parkas des travaux conduits par Albert Cohen rapporteur de cette thèse) et ses collaborateurs visent à exprimer des bibliothèques spécifiques de calcul en utilisant un format interne de représentation qui se compile vers C++ ou OpenCL, et permet de les inclure dans des codes de plus large échelle [43]. Des plates-formes comme StarPU ou XKaapi permettent également de modéliser des applications par des graphes de tâches pour étudier spécifiquement leur placement-ordonnancement sur des architectures de nature CPU-GPU (de manière dynamique, mais sur des graphes acycliques)[44] [45].

Les architectures de nature MP-SoC, ou MPPA (*Massively Parallel Processor Array*) peuvent aussi être des cibles de choix pour une implantation de notre nature (trouver le bon niveau de cœur pour le déploiement d'une FFT unique, puis combiner les ressources pour exécuter de nombreuses FFT en parallèle suivant les besoins de l'application sur les différentes ressources de l'architecture), si ces types de circuits se développent au niveau commercial chez les clients de Kontron.

Les prochaines architectures *Intel Core* de sixième génération *skylake*, annoncent l'introduction dans leur jeu d'instructions [7] de nouvelles unités SIMD de largueur 512bits (AVX-512). Ces dernières incluent un masque d'exécution sur le SIMD qui permettra à terme de pouvoir utiliser le modèle SIMT sur CPU, jusque là adopté par les GPU. Nous pensons aussi que ces améliorations architecturales ouvrent de nouvelles perspectives vers un modèle de programmation fédérateur.

Les progrès en fréquence de fonctionnement des technologies silicium, pour bâtir des architectures de calcul intensif, sont maintenant limités depuis plusieurs années. La voie naturelle suivie pour l'accroissement des performances, en tirant partie des gravures toujours plus fines, est l'augmentation du nombre de cœurs généralistes (CPUs) et du nombre de cœurs optimisés pour le calcul (GPUs). Et le langage associé à cette parallélisation des cœurs est l'OpenCL. Ce formalisme devra continuer de se perfectionner, et de jouer d'une part, le rôle de passerelle entre les multi-cœurs CPU et les multi-cœurs GPU, et d'autre part, le rôle d'agent de compatibilité indispensable entre les micro-architectures binairement différentes d'un vendeur à l'autre, et d'une génération GPU à l'autre.

6. https://team.inria.fr/aoste/
7. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf

Bibliographie

[1] M. Frigo and S. Johnson, “The Design and Implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, pp. 216 –231, Feb. 2005.

[2] B. Cipra, “The Best of the 20th Century : Editors Name Top 10 Algorithms,” *SIAM News*, vol. 33, no. 4, 2000.

[3] J. R. Johnson and R. W. Johnson, “Challenges of Computing the Fast Fourier Transform,” in *IN DARPA CONFERENCE*, 1997.

[4] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.*, vol. 19, pp. 297–301, 1965.

[5] E. Quinnell, E. Swartzlander, and C. Lemonds, “Floating-Point Fused Multiply-Add Architectures,” in *Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers, 2007. ACSSC 2007*, pp. 331–337, Nov. 2007.

[6] E. Linzer and E. Feig, “Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures,” *IEEE Transactions on Signal Processing*, vol. 41, pp. 93–, Jan. 1993.

[7] E. Linzer and E. Feig, “Modified FFTs for Fused Multiply-Add Architectures,” *Mathematics of Computation*, vol. 60, pp. 347–361, Jan. 1993.

[8] S. Goedecker, “Fast radix 2, 3, 4, and 5 kernels for fast fourier transformations on computers with overlapping multiply-add instructions,” *SIAM J. Sci. Comput.*, vol. 18, pp. 1605–1611, Nov. 1997.

[9] J.-P. Perez-Seva, *The optimizations of signal processing algorithms of modern parallel and embedded architectures.* Theses, Université Nice Sophia Antipolis, Aug. 2009.

[10] M. A. Bergach, S. Tissot, M. Syska, and R. De Simone, “Scaling Performance of FFT Computation on an Industrial Integrated GPU Co-processor : Experiments with Algorithm Adaptation,” (Dresden, Germany), ECSI, Mar. 2014.

[11] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, Jan. 1992.

[12] W. Cochran, J. W. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, J. Maling, G.C., D. Nelson, C. Rader, and P. D. Welch, “What is the fast Fourier transform ?,” *Proceedings of the IEEE*, vol. 55, pp. 1664–1674, Oct. 1967.

[13] R. C. Singleton, “An algorithm for computing the mixed radix fast Fourier transform,” *IEEE Transactions on Audio and Electroacoustics*, vol. 17, no. 2, pp. 93–103, 1969.

[14] P. Duhamel, “Un algorithme de transformation de Fourier rapide à double base,” *Annales des Télécommunications*, vol. 40, pp. 481–494, Sept. 1985.

[15] S. G. Johnson and M. Frigo, “A Modified Split-Radix FFT With Fewer Arithmetic Operations,” *Signal Processing, IEEE Transactions on*, vol. 55, pp. 111 –119, Jan. 2007.

[16] D. J. Bernstein, "The Tangent FFT," in *Proceedings of the 17th International Conference on Applied Algebra, Algebraic Algorithms and Error-correcting Codes*, AAECC'07, (Berlin, Heidelberg), pp. 291–300, Springer-Verlag, 2007.

[17] I. J. Good, "The interaction algorithm and practical Fourier analysis," *Journal of the Royal Statistical Society. Series B*, 1960.

[18] P. Duhamel and M. Vetterli, "Fast fourier transforms : A tutorial review and a state of the art," *Signal Processing*, vol. 19, pp. 259–299, Apr. 1990.

[19] S. Winograd, "On Computing the Discrete Fourier Transform," *Mathematics of Computation*, vol. 32, p. 175, Jan. 1978.

[20] S. Winograd, "On the multiplicative complexity of the Discrete Fourier Transform," *Advances in Mathematics*, vol. 32, pp. 83–117, May 1979.

[21] T. Hartley, A. Fasih, C. Berdanier, F. Ozguner, and U. Catalyurek, "Investigating the use of GPU-accelerated nodes for SAR image formation," in *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09*, pp. 1–8, 2009.

[22] O. Altun, S. Paker, and M. Kartal, "Realization of interpolation-free fast sar range-doppler algorithm using parallel processing on gpu," in *Progress In Electromagnetics Research Symposium Proceedings*, pp. 998–1002, PIERS, 2013.

[23] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining register allocation and instruction scheduling," tech. rep., Stanford, CA, USA, 1995.

[24] D. Koufaty and D. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, pp. 56–65, Mar. 2003.

[25] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition : A Quantitative Approach.* Waltham, MA : Morgan Kaufmann, 5 edition ed., Sept. 2011.

[27] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 130–141, ACM, 2013.

[28] F. Franchetti and M. Püschel, *Encyclopedia of Parallel Computing*, ch. Fast Fourier Transform. Springer, 2011.

[29] C. Temperton, "Self-Sorting In-Place Fast Fourier Transforms," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, pp. 808–823, July 1991.

[30] M. Frigo and S. G. Johnson, "Fftw : An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384, IEEE, 1998.

[31] D. Mirković and S. L. Johnsson, "Automatic performance tuning in the uhfft library," in *Computational Science ICCS 2001*, pp. 71–80, Springer, 2001.

[32] E. W. Dijkstra, *A short introduction to the art of programming*, vol. 4. Technische Hogeschool Eindhoven Eindhoven, 1971.

[33] A. M. Blake, *Computing the fast Fourier transform on SIMD microprocessors.* Thesis, University of Waikato, 2012.

[34] S. Ocovaj and Z. Lukac, “Optimization of conjugate-pair split-radix FFT algorithm for SIMD platforms,” in *2014 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 373–374, Jan. 2014.

[35] W. Xu, Z. Yan, and D. Shunying, “A high performance FFT library with single instruction multiple data (SIMD) architecture,” in *2011 International Conference on Electronics, Communications and Control (ICECC)*, pp. 630–633, Sept. 2011.

[36] K. Zhang, S. Chen, S. Liu, Y. Wang, and J. Huang, “Accelerating the data shuffle operations for FFT algorithms on SIMD DSPs,” in *2011 IEEE 9th International Conference on ASIC (ASICON)*, pp. 683–686, Oct. 2011.

[37] H. Izumi, K. Sasaki, K. Nakajima, and H. Sato, “An Efficient Technique for Corner-Turn in SAR Image Reconstruction by Improving Cache Access,” in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS ’02, (Washington, DC, USA), pp. 67–, IEEE Computer Society, 2002.

[38] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, “High performance discrete Fourier transforms on graphics processors,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC ’08, (Piscataway, NJ, USA), pp. 2 :1–2 :12, IEEE Press, 2008.

[39] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, “An efficient, model-based CPU-GPU heterogeneous FFT library,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1 –10, Apr. 2008.

[40] V. Volkov and B. Kazian, “Fitting fft onto the g80 architecture,” *University of California, Berkeley*, 2008.

[41] M. Daga, A. M. Aji, and W.-c. Feng, “On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing,” in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC ’11, (Washington, DC, USA), pp. 141–149, IEEE Computer Society, 2011.

[42] V. Kelefouras, G. Athanasiou, N. Alachiotis, H. Michail, A. Kritikakou, and C. Goutis, “A Methodology for Speeding Up Fast Fourier Transform Focusing on Memory Architecture Utilization,” *IEEE Transactions on Signal Processing*, vol. 59, no. 12, pp. 6217–6226, 2011.

[43] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. Donaldson, “Pencil : Towards a platform-neutral compute intermediate language for dsls,” in *2nd Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC, associated with SC)*, (Salt Lake City, Utah), Nov. 2012.

[44] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu : A unified platform for task scheduling on heterogeneous multicore architectures,” *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, Feb. 2011.

[45] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, “Xkaapi : A runtime system for dataflow task programming on heterogeneous architectures,” in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS ’13, (Washington, DC, USA), pp. 1299–1308, IEEE Computer Society, 2013.