Adéquation Algorithme Architecture et modèle de programmation pour l'implémentation d'algorithmes de traitement du signal et de l'image sur cluster multi-GPU

Vincent Boulos

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Signal, Image, Parole, Télécoms**

Arrêté ministériel : 7 août 2006

Présentée par

Vincent BOULOS

Thèse dirigée par **Dominique HOUZET**
codirigée par **Luc SALVO**
et encadrée par **Vincent FRISTOT** et **Sylvain HUET**

préparée au sein **du laboratoire GIPSA-lab**
et de **l'Ecole Doctorale Electronique, Electrotechnique, Automatique & Traitement du Signal**

Adéquation Algorithme Architecture et modèle de programmation pour l'implémentation d'algorithmes de traitement d'images 2D-3D sur cluster multi-GPU

Thèse soutenue publiquement le **18 décembre 2012**,
devant le jury composé de :

Mr, Pierre-Yves COULON
PR à Grenoble-INP, Président
Mr, Dominique BERNARD
DR à ICMCB-CNRS, Bordeaux, Rapporteur
Mr, Lionel LACASSAGNE
MCF HDR à IEF Paris Sud, Rapporteur
Mr, Jean-Pierre BRUANDET
Docteur et Directeur Technique à Digisens, Chambéry, Examinateur
Mr, Maxime PELCAT
MCF à INSA de Rennes, Examinateur
Mr, Samuel THIBAULT
MCF à Université de Bordeaux 1, Examinateur
Mr, Dominique HOUZET
PR à Grenoble-INP, Directeur de thèse
Mr, Luc SALVO
PR à Grenoble-INP, Co-Directeur de thèse

Résumé

Initialement conçu pour décharger le CPU des tâches de rendu graphique, le GPU est devenu une architecture massivement parallèle adaptée au traitement de données volumineuses. Alors qu'il occupe une part de marché importante dans le Calcul Haute Performance, une démarche d'Adéquation Algorithme Architecture est néanmoins requise pour implémenter efficacement un algorithme sur GPU.

La contribution de cette thèse est double. Dans un premier temps, nous présentons le gain significatif apporté par l'implémentation optimisée d'un algorithme de granulométrie (l'ordre de grandeur passe de l'heure à la minute pour un volume de 1024^3 voxels). Un modèle analytique permettant d'établir les variations de performance de l'application de granulométrie sur GPU a également été défini et pourrait être étendu à d'autres algorithmes réguliers.

Dans un second temps, un outil facilitant le déploiement d'applications de Traitement du Signal et de l'Image sur cluster multi-GPU a été développé. Pour cela, le champ d'action du programmeur est réduit au découpage du programme en tâches et à leur mapping sur les éléments de calcul (GPP ou GPU). L'amélioration notable du débit sortant d'une application streaming de calcul de carte de saillence visuelle a démontré l'efficacité de notre outil pour l'implémentation d'une solution sur cluster multi-GPU. Afin de permettre un équilibrage de charge dynamique, une méthode de migration de tâches a également été incorporée à l'outil.

Mots clés - Adéquation Algorithme Architecture, GPGPU, modèle de programmation, architecture hétérogène, multi-GPU

Abstract

Originally designed to relieve the CPU from graphics rendering tasks, the GPU has become a massively parallel architecture suitable for processing large amounts of data. While it has won a significant market share in the High Performance Computing domain, an Algorithm-Architecture Matching approach is still necessary to efficiently implement an algorithm on GPU.

The contribution of this thesis is twofold. Firstly, we present the significant gain provided by the implementation of a granulometry optimized algorithm (computation time decreases from several hours to less than minute for a volume of 1024^3 voxels). An analytical model establishing the performance variations of the granulometry application is also presented. We believe it can be expanded to other regular algorithms.

Secondly, the deployment of Signal and Image processing applications on multi-GPU cluster can be a tedious task for the programmer. In order to help him, we developped a library that reduces the scope of the programmer's contribution in the development. His remaining tasks are decomposing the application into a Data Flow Graph and giving mapping annotations in order for the tool to automatically dispatch tasks on the processing elements (GPP or GPU). The throughput of a visual sailency streaming application is then improved thanks to the efficient implementation brought by our tool on a multi-GPU cluster. In order to permit dynamic load balancing, a task migration method has also been incorporated into it.

Keywords - Algorithm Architecture Matching, GPGPU, programming model, heterogeneous architecture, multi-GPU

Table des matières

1 Remerciements **1**

2 Introduction **3**
2.1 A l'apogée du modèle séquentiel 3
2.2 Contributions de cette thèse 6
2.3 Contexte de travail 6
2.4 Organisation du manuscrit 10

I Parallélisme à grain fin : Adéquation Algorithme Architecture d'une application de traitement de l'image sur GPU **13**

3 General-Purpose computing on GPU **15**
3.1 Les types d'architectures 16
3.2 Les accélérateurs 18
3.3 Le GPGPU appliqué au traitement de l'image 24
3.4 Les langages de programmation pour le GPGPU 28
3.5 Le paradigme de programmation CUDA 30
3.6 Les "paralléliseurs" pour GPU 38

4 Accélération de l'application de granulométrie **41**
4.1 La granulométrie 42
4.2 Travaux existants 44
4.3 Optimisations de l'algorithme de granulométrie 45
4.4 Performances crêtes et facteur limitant sur GPU 52
4.5 Implémentation de l'algorithme optimisé sur GPU 54
4.6 Résultats expérimentaux 72
4.7 Conclusion 88

II Parallélisme à gros grain : déploiement d'applications de traitement du signal et de l'image sur cluster multi-GPU **91**

5 Modèles de programmation parallèle **93**
5.1 Les types de parallélisme 93

5.2 Les types d'architectures parallèles 96
5.3 Les modèles de programmation pour architectures parallèles 97
5.4 Conclusion 103

6 PACCOlib : une librairie pour cluster multi-GPU 105
6.1 Notre flot de conception 106
6.2 Cas d'étude 117
6.3 Conclusion 129

7 Migration de tâches 135
7.1 Motivations 135
7.2 Implémentation 136
7.3 Résultats 141
7.4 Conclusion 145

8 Conclusion 147

A Bases de morphologie mathématique 151

B Caractéristiques des GPUs 155
B.1 Caractéristiques générales 155
B.2 Accès coalesced à la mémoire globale 155

C Code des différentes versions de kernels testés avec le profiler 159

Chapitre 1

Remerciements

Je tiens à remercier mon directeur de thèse Pr. Dominique HOUZET et mes encadrants Dr. Vincent FRISTOT, Dr. Sylvain HUET et Dr. Luc SALVO pour leur disponibilité et la guidance qu'ils ont pu m'apporter durant ces trois années de recherche. Merci pour votre disponibilité dans ces derniers jours de préparation avant la soutenance.

Merci à mes co-bureaux passés et présents : Jonathan PINEL pour son expertise en Latex, ses cookies et ses tartes revigorantes, Lionel BOMBRUN pour son rire communicatif et ses talents de coincheur invétéré, Yun Jie WU pour les dégustations et ses cours de chinois, Mohammed JABAR pour son calme et sa sérénité, Aude COSTARD pour sa bonne humeur, son sourire ravageur et sa sensibilité face au moral des troupes.

Je tiens également à remercier tous les doctorants du DIS et tout particulièrement Matthieu SANQUER pour l'initiation à la coinche et à bon nombre de jeux de sociétés, Jérémie BOULANGER pour ses réfléxions mathématiques appliquées à la vie de tous les jours, Wei FAN pour son bon karma, Gailene PHUA pour ses émoticones, Robin GERZAGUET pour sa coolatitude, Flore HARLE pour les délirs et tout ceux avec qui j'ai pu partagé les mets délicats et raffinés du RU : Cédric ROUSSET, Bastien LYONNET, Cyrille MIGNOT et j'en passe.

Merci au GipsADoc, l'association des doctorants du Gipsa-Lab, d'avoir animé nos vies de doctorants et au laboratoire Gipsa-Lab pour les locaux mis à notre disposition.

Merci aux permanents, à notre secrétaire Lucia BOUFFARD-TOCAT et aux membres administratifs.

Finalement, je souhaite remercier ma famille et ma fiancée pour leur soutien et leur aide dans ces derniers mois de rédaction. -

Chapitre 2

Introduction

Sommaire

2.1 A l'apogée du modèle séquentiel 3
2.2 Contributions de cette thèse 6
2.3 Contexte de travail 6
2.3.1 Choix de l'application 7
2.3.2 Choix de l'architecture cible 8
2.3.3 Un modèle de programmation pour le déploiement d'applications de Traitement du Signal et l'Image sur cluster multi-GPU 10
2.4 Organisation du manuscrit 10

2.1 A l'apogée du modèle séquentiel

Nous sommes en ce XXIe siècle dans une période charnière de l'histoire de l'informatique. Les architectures parallèles longtemps délaissées au détriment des architectures séquentielles à cause de la complexité de leur programmation émergent enfin. Les limites physiques poussent les fabricants de processeurs à orienter leur production vers une multiplication du nombre de cœurs. Pour cela, la terminologie *Central Processing Unit* (CPU) faisant référence à une unité de calcul centrale unique n'est plus adaptée. Nous parlons alors de *General Purpose Processor* (GPP). La course aux performances menée par cette nouvelle lignée de processeurs multi-cœurs a été rejointe par les processeurs graphiques (GPU pour Graphics Processing Unit) et leur puissance de calcul phénoménale (Figure 2.1). En effet, depuis 2004, les GPU ont élargi leur domaine d'application en permettant leur exploitation pour effectuer du calcul scientifique. Cette nouvelle pratique est connue sous la terminologie *General-Purpose computations on GPU* (GPGPU).

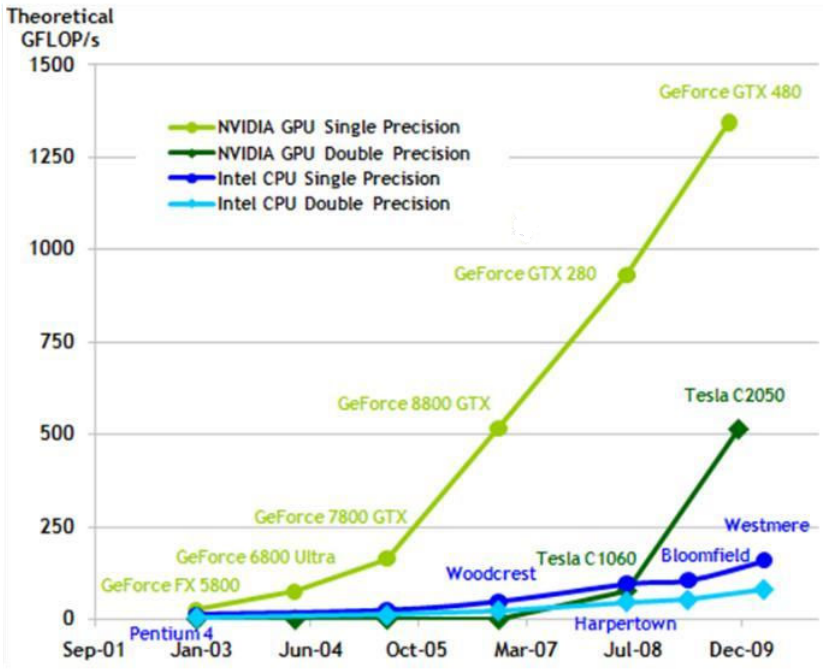
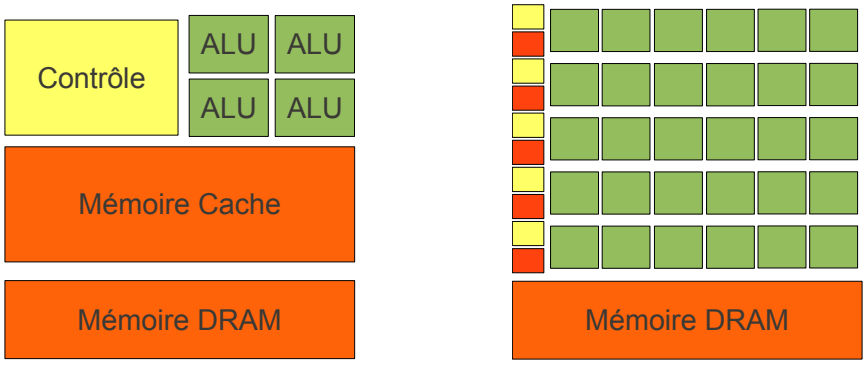


FIGURE 2.1 – Evolution de la puissance de calcul des GPU en GFLOPS comparée à celle des CPU au cours des dernières années.

La programmation scientifique sur GPU a commencé à se démocratiser alors que les paradigmes de programmation existants étaient encore figés sur le pipeline graphique. Le premier cluster de GPU [36], constitué de 32 nœuds, a été programmé en Cg [73], un langage graphique. Depuis, le pipeline graphique des GPU a fusionné et leur architecture est désormais une architecture parallèle à part entière. En outre, maintenant que des paradigmes de programmation tels que *Compute Unified Device Architecture* (CUDA) et *Open Computing Language* (OpenCL) existent, une réelle effervescence tourne autour du GPGPU devenu accessible à tous.

Le GPP et le GPU ont pénétré l'ère du parallélisme quasi-simultanément. Leur approche reste cependant distincte. En effet, ils conservent leurs propres spécificités mais tous deux évoluent pour tendre vers un modèle d'architecture qui s'approche davantage de l'autre. La différence majeure entre GPP et GPU est dans la spécification des transistors (Figure 2.2). Tandis qu'un GPP dédie une partie importante de ses transistors aux niveaux de mémoire cache, un GPU les exploite essentiellement en tant qu'*Unité Arithmétique et Logique* (ALU pour Arithmetic and Logical Unit). Ainsi, le GPP gère mieux les tâches imprévisibles et éventuellement lourdes nécessitant une gestion complexe des données (un système d'exploitation par exemple). Par contre, le GPU, mieux équipé pour traiter une quantité massive de données, est plus adapté à l'accélération de tâches régulières (traitement de l'image, encodage, etc).

Derrière la profonde remise en question et la refonte des architectures GPP et GPU pour répondre au mieux aux exigences du siècle actuel, alors que la pédagogie informatique est



(a) GPP (b) GPU

FIGURE 2.2 – Comparaison de la répartition fonctionnelle des transistors sur GPP vs GPU.

ancrée dans un paradigme de programmation séquentiel, nous sommes confrontés à la nécessité d'évoluer vers un modèle de programmation parallèle. Cela constitue un réel défi pour la nouvelle vague de programmeurs diplômés qui débarquent sur le marché du travail à l'ère du parallèle. En effet, les modèles de programmation mis à la disposition des développeurs ne sont pas adaptés à l'exploitation des architectures sous-jacente. Alors que les fabricants de processeurs (essentiellement GPP et GPU, même si tout le secteur des machines de calculs est touché) tâtent encore activement le terrain à la recherche de la recette de composants miracles pour créer la référence du multi-processeur multi-cœur de demain (comme l'Intel x86 l'a été jusque là), les développeurs adaptent tant bien que mal les outils qui sont mis à leur disposition pour l'exploitation d'architectures aux caractéristiques variables et en pleine mouvance.

A l'ère du séquentiel, il était possible de simuler la concurrence (couche logicielle) sur une architecture à processeur unique (couche matérielle) grâce aux threads. A l'heure actuelle, les modèles de programmation les plus répandus pour la création et la gestion des threads sont : les *threads POSIX* (Pthreads) et *Message Passing Interface* (MPI) [1] pour la communication entre les tâches et OpenMP [2] pour l'expression du parallélisme de données (OpenMP permet également un certain parallélisme de tâches). Grâce aux mécanismes de synchronisation, le programmeur peut réguler le trafic des données et attribuer les priorités d'accès entre threads. Néanmoins, cette gestion des threads à grain fin, devient rapidement un casse-tête pour l'être humain à cause des myriades d'interactions qu'il doit gérer manuellement au niveau de la mémoire. Ces heurts, implicites (lectures/écritures dans la même zone mémoire) ou explicites (transmissions de variables), rendent la programmation complexe et rapidement confuse.

On peut distinguer **deux niveaux d'expression du parallélisme** : le **parallélisme à gros grain** et le **parallélisme à grain fin**. D'une part, le parallélisme matériel inhérent

1. www.mcs.anl.gov/mpi/index.htm
2. http ://openmp.org

aux architectures parallèles émergentes et, d'autre part, le parallélisme *multi-threading* connu jusque là sur les architectures séquentielles. Toutefois, l'écart entre ces deux niveaux de parallélisme est conséquent. Sans passer par ces deux extrêmes, il manque d'élever le niveau d'abstraction de la programmation parallèle sur les architectures de calcul à un grain moyen.

Dans cette optique, la démarche qu'a entamé Nvidia en combinant les parallélismes logiciel et matériel dans son paradigme de programmation CUDA est intéressante. En effet, malgré une architecture massivement parallèle et plusieurs milliers de threads, le degré d'ingérence du programmeur dans les interactions entre les threads est réduite grâce au regroupement des threads en blocs sans pour autant le couper de la possibilité de gérer les threads individuellement (les détails de ce paradigme de programmation seront discutés dans le Chapitre 3).

2.2 Contributions de cette thèse

La contribution de cette thèse est double.

- Dans un premier temps, nous nous intéresserons à accélérer un algorithme de traitement d'images très utilisé dans le domaine de la science des matériaux : la granulométrie (Chapitre 4). Dans cette partie, nous abordons le problème sous un angle de **parallélisme à grain moyen-fin** grâce aux paradigmes de programmation CUDA et OpenMP.
- Dans un deuxième temps, nous présenterons la librairie *Parallel Computation-Communication Overlap library* (PaCCOlib) (Chapitre 6) développée dans le but de simplifier le déploiement d'applications de traitement d'images 2D-3D sur cluster hybride GPP-GPU. Dans cette partie, nous aborderons le problème sous un angle de **parallélisme à gros grain** grâce aux communications inter-processeurs via les langages de programmation MPI et les Pthreads.

2.3 Contexte de travail

Étant donné les besoins grandissant au laboratoire *Science et Ingénierie des Matériaux et Procédés* (SIMaP) d'accélérer les traitements d'images sur des données de plus en plus volumineuses, les laboratoires *Grenoble Image Parole Signal Automatique* (GIPSA-lab) et SIMaP ont choisi d'allier leur expertise respective : l'adéquation algorithme architecture et l'application de la tomographie dans l'étude des propriétés des matériaux, afin d'accélérer le traitement d'images volumineuses sur architecture parallèle. Pour cela, deux choix spécifiant le cadre de travail plus en détail ont dû se faire : le choix de l'application a accélérer (2.3.1) et le choix de l'architecture (2.3.2).

2.3.1 Choix de l'application

Depuis le début des années 2000, la tomographie connaît un essor considérable pour l'étude des caractéristiques des matériaux. Cette technique d'imagerie permet une acquisition rapide à haute résolution d'un volume 3D issu de la projection de rayons X traversant le matériau étudié. L'utilisation de la tomographie dans le domaine de la science des matériaux s'effectue en trois étapes successives :

L'acquisition des projections (au synchrotron ou avec des tomographes de laboratoires). A l'*European Synchrotron Radiation Facility* (ESRF), les temps d'acquisition des projections sont inférieurs à 1 seconde pour un volume de 1024^3 voxels.

La reconstruction du volume 3D à partir des projections. Désormais, elle se fait systématiquement à l'aide de GPU. Grossièrement, la reconstruction d'une image 1024^3 prends moins de 5 minutes.

Le traitement d'image appliqué au volume 3D en vue d'interprétations physiques. Cette étape étant de loin l'étape la plus longue, des progrès sur la durée de traitement des volumes 3D est nécessaire. La chaîne classique des traitements post-tomographique d'une image est résumée au Tableau 2.1. Parmi ceux-ci, notre but est de déceler le traitement le plus intéressant à accélérer. Pour cela, nous nous baserons sur deux facteurs : le temps d'exécution et l'utilité (usage systématique ou ponctuel).

1- Récupération du volume en niveau de gris

2- Filtrage Généralement, avec un filtre médian de taille 1 (algorithme de fenêtre glissante), il faut compter environ 5 minutes sur un volume 1024^3. Il existe des filtres plus sophistiqués, assez puissant, comme la diffusion anisotrope 3D qui sont très longs sur GPP mais une implémentation GPU a été mise en place dans le logiciel Avizo Fire.

3- Segmentation Une implémentation GPU dans ImageJ exécute cette opération en moins de 5 secondes.

A1.1- Séparation des objets Cette étape potentiellement longue (environ 15 minutes sur un volume 1024^3) n'est pas toujours nécessaire.

A1.2- Labellisation Cela dépendra du nombre d'objets et de leur taille. Généralement, il faut compter entre 1 à 2 minutes pour un volume 1024^3.

A1.3- Calcul des paramètres Cela dépendra du nombre de paramètres à calculer. Généralement, il faut compter 5 minutes sur un volume 1024^3 pour une trentaine de paramètres.

A2.1- Granulométrie Pour extraire la taille des pores et l'épaisseur de la structure des mousses métalliques [117, 20], une granulométrie réalisée avec le logiciel imorph [20] sur un volume de dimensions 600 x 600 x 250 avec des objets de taille maximale 50 pixels requiert 15 minutes pour une implémentation GPP (cette durée peut être réduite en sacrifiant sur la précision des résultats [2]). Cependant, alors que l'acquisition de larges quantités de données est désormais très rapide (moins d'une

Traitement	Temps de calcul
1- Récupération du volume en niveau de gris	$<$ 1 seconde
2- Filtrage	5 minutes
3- Segmentation	$<$ 5 secondes
Analyse 1	
A1.1- Séparation des objets	15 minutes
A1.2- Labellisation	1-2 minutes
A1.3- Calcul des paramètres	5 minutes
Analyse 2	
A2.1- Granulométrie	plusieurs heures

TABLE 2.1 – Chaîne classique des traitements post-tomographiques sur un volume 3D de 1024^3 voxels.

seconde pour acquérir un volume de 1024^3 [101]), le traitement de tels volumes nécessite plusieurs heures de calcul [105, 109] avec des logiciels comme Aphélion, très utilisé dans le domaine.

Pour conclure, dans la chaîne d'analyse le point faible est très clairement la granulométrie : sur un volume 1024^3, le temps d'exécution des autres traitements est de l'ordre de la dizaine de minutes maximum tandis qu'il occupe plusieurs heures pour une granulométrie. D'où le choix de porter cette application.

2.3.2 Choix de l'architecture cible

Suite aux travaux d'un doctorant au laboratoire GIPSA-lab [41], une implémentation d'un algorithme de reconstruction 3D à partir d'images médicales acquises par tomographie a été testée sur diverses architectures cibles : GPP, FPGA et GPU. Les performances du GPU (Tableau 2.2) se sont révélées intéressantes avec des performances du même ordre de grandeur qu'un ASIC et une programmation rapide et simple (comparée aux autres solutions). Pour cela, le GPU est l'architecture parallèle qui a été retenu pour l'accélération d'algorithmes durant cette thèse.

	Hardware	Nb Unités de Traitement	Temps	Cycles/Op /UT	Cycles/Op total
GPP	Pentium 4 (3.2 Ghz, 6.4 Go/s)	1	2.5 s	16	16
	bi-Xeon dual core (3.0 Ghz, 10.6 Go/s)	4	294 ms	7.12 s	1.78
GPU	GTS8800 (1.2 Ghz, 64 Go/s)	96	50 ms	12.9	0.135
FPGA	Virtex 4 (0.2 MGhz, 0.8 Go/s)	8	526 ms	1,7	0,21
ASIC	5*3PA-PET (1.2 Ghz, 24 Go/s)	40	27 ms	2.62	0.065

TABLE 2.2 – Temps de reconstruction sur GPP / FPGA / GPU [41].

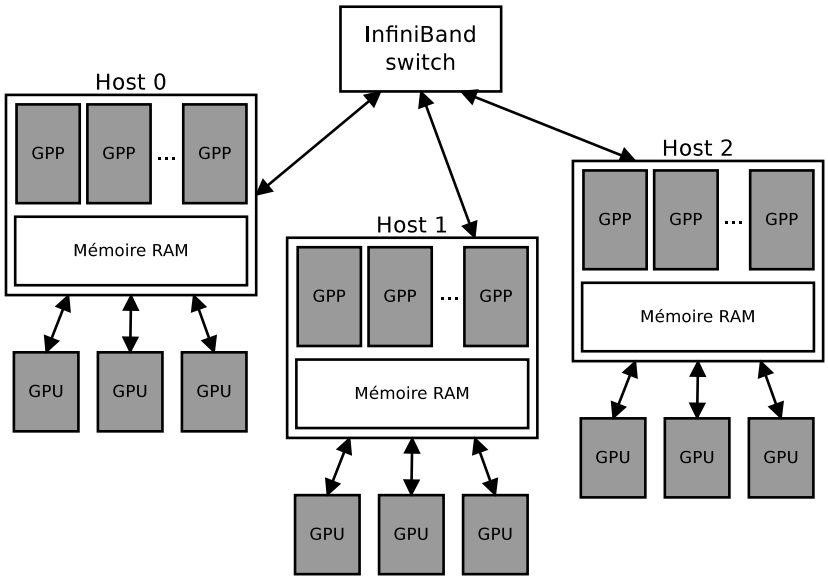


FIGURE 2.3 – Cluster hybride GPP-GPU mis en place par l'équipe AGPIG du laboratoire GIPSA-lab. Les composants de couleur grise constituent les éléments de calcul.

2.3.3 Un modèle de programmation pour le déploiement d'applications de Traitement du Signal et l'Image sur cluster multi-GPU

Une fois familiarisé au GPGPU, il est dans la continuité d'envisager les gains potentiels d'une solution multi-GPU. L'équipe AGPIG possède au sein du laboratoire GIPSA-lab un cluster de calcul composé de trois nœuds GPP connectés par InfiniBand et contenant chacun plusieurs cartes GPU (Figure 2.3). L'exploitation de cette architecture hétérogène est néanmoins complexe à cause de la multiplicité des langages de programmation qui peuvent être utilisés :

- C, C++, JAVA, etc pour le GPP,
- CUDA ou OpenCL pour le GPU,
- les pthreads et MPI pour les communications entre les éléments de calculs.

Pour cette raison, PaCCOlib, un modèle de programmation simplifiant le déploiement d'applications de traitement d'images sur cluster multi-GPU, a été développé. Cet outil est capable d'abstraire la création et la gestion des threads, l'allocation et les transferts mémoire entre les éléments de calculs ainsi que le recouvrement des opérations de calcul et de communication. Afin de permettre au programmeur d'utiliser les ressources matérielles du cluster de façon efficace, une méthode de migration de tâches a également été incorporée à l'outil.

2.4 Organisation du manuscrit

Le manuscrit est constitué de deux parties :

Partie I : Parallélisme à grain fin

Les architectures parallèles pour l'accélération des traitements sur GPP Le Chapitre 3 reprendra l'évolution des calculateurs avant de présenter les architectures parallèles communément répandus pour accélérer les calculs. Parmi ces accélérateurs, nous nous attarderons plus particulièrement sur le GPU, son évolution depuis son usage typiquement graphique à son accueil fortement sollicité dans la communauté scientifique. Cette entrée en matière sera accompagnée d'une description détaillée du paradigme de programmation CUDA suivie d'une liste des modèles de programmation simplifiant le portage de codes séquentiels pour des personnes non-initiées au GPGPU.

Adéquation Algorithme Architecture de la granulométrie sur GPU Le Chapitre 4 sera consacré à l'accélération de l'algorithme de granulométrie sur GPU. Pour cela, nous optimiserons progressivement l'algorithme de granulométrie avant de détailler les étapes de son implémentation sur GPU. Finalement, à partir de résultats expérimentaux, nous tenterons d'établir un lien entre les performances de l'implémentation GPU, le paradigme de programmation CUDA et la taille du volume fourni en entrée.

Partie II : Parallélisme à gros fin

Les modèles de programmation pour clusters Dans le Chapitre 5, nous classifierons les modèles de programmation pour architectures hétérogènes GPP-GPU en fonction de plusieurs caractéristiques qui y serons définies. Ce chapitre nous permettra d'établir un état de l'art autour des modèles de programmation pour cluster afin de repérer les axes de recherche abordés, saturés et à venir et de positionner notre libraire PaCCOlib parmi eux.

PaCCOlib : une librairie pour le déploiement d'applications sur cluster Dans le Chapitre 6, nous présentons PaCCOlib, un modèle de programmation pour le déploiement d'applications streaming sur cluster multi-GPU. Ce modèle de programmation orienté Data-Flow permet de faciliter considérablement la programmation sur cluster en faisant abstraction au développeur de toutes les routines de codage de communications inter-processeurs.

Migration de tâches Dans le Chapitre 7, la méthode de migration de tâche ajoutée à notre librairie PACCOlib est présentée. Cette méthode apporte davantage de flexibilité à la librairie PACCOlib. Elle permet la modification en ligne du mapping des tâches sur les éléments de calcul. L'approche est décomposée en plusieurs actions exécutées successivement à chaque itération du cycle de fonctionnement de l'application. Ainsi, la perturbation introduite dans le cycle de fonctionnement est réduit.

Première partie

Parallélisme à grain fin : Adéquation Algorithme Architecture d'une application de traitement de l'image sur GPU

Chapitre 3

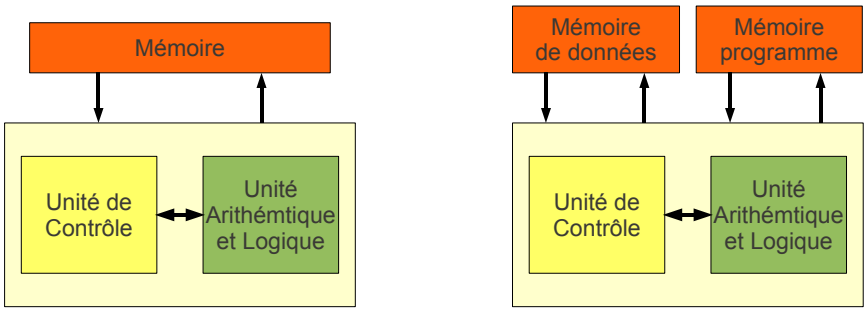
General-Purpose computing on GPU

Sommaire

3.1 Les types d'architectures 16
3.2 Les accélérateurs 18
3.2.1 Le Field Programmable Gate Array (FPGA) 21
3.2.2 Le Cell Broadband Engine (Cell BE) 22
3.2.3 Le Graphics Processing Unit (GPU) 23
3.3 Le GPGPU appliqué au traitement de l'image 24
3.4 Les langages de programmation pour le GPGPU 28
3.5 Le paradigme de programmation CUDA 30
3.5.1 L'architecture 30
3.5.2 Le modèle de programmation 33
3.5.3 Les goulots d'étranglement dans la programmation GPGPU . 35
3.6 Les "paralléliseurs" pour GPU 38

« Les GPU ont tellement évolué que de nombreuses applications professionnelles internationales utilisent aujourd'hui leur puissance de calcul pour s'exécuter bien plus vite qu'avec un système multi-cœurs standard. Les architectures de calcul du futur seront des systèmes hybrides exploitant à la fois les GPU parallèles et les CPU multi-cœurs. » Prof. Jack Dongarra, Directeur de l'Innovative Computing Laboratory, Université du Tennessee.

Dans ce chapitre, des notions d'architecture et d'histoire des calculateurs seront abordées afin d'expliquer comment nous en somme venu au *General-Purpose computing on GPU*. Dans la section 3.1, nous présenterons la classification des architectures d'ordinateur. Dans la section 3.2, nous présenterons les évènements à l'origine du tournant que subit actuellement



(a) Architecture de Von Neumann.

(b) Architecture de Harvard.

FIGURE 3.1 – Les deux principales architectures de processeurs.

l'informatique. Dans la section 3.3, nous défendrons le choix du GPU pour l'accélération d'applications de traitement d'image. Dans la section 3.4, nous présenterons les langages de programmation pour le GPGPU. Dans la section 3.5, nous détaillerons le paradigme de programmation CUDA. Dans la section 3.6, nous passerons en revue les solutions existantes facilitant le portage d'algorithmes séquentiels sur accélérateurs GPU.

3.1 Les types d'architectures

Un ordinateur est composé de deux parties : le logiciel et le matériel. Le matériel est exploité grâce au logiciel. Ce support logiciel peut être simple (une machine d'état comme celle que l'on peut trouver dans l'ordinateur de bord d'un ascenceur, par exemple) ou complexe (un logiciel de simulation numérique, par exemple). Afin d'exploiter le matériel, le travail du programmeur est donc double : (1) formaliser la solution qu'il souhaite implémenter (2) dans un langage compréhensible par la machine. Chaque architecture détient son propre jeu d'instructions (ISA pour Instruction Set Architecture) permettant d'exploiter le matériel. Celui-ci est généralement constitué d'au moins trois composants : la mémoire pour le stockage des données et des instructions du programme, l'unité de contrôle chargée du séquençage des données et l'*Unité Arithmétique et Logique* (ALU pour Arithmetic and Logical Unit) pour les calculs. Des dispositifs d'entrée/sortie sont généralement présents pour communiquer avec l'extérieur. L'assembleur est le langage de programmation le plus bas niveau qui puisse exister. Il permet de programmer le matériel dans les moindres détails. Cette tâche peut s'avérer complexe pour des programmes long et une architecture vaste. Pour cela, des langages de programmation haut niveau traduits en langage machine par un compilateur ont été inventés. Ils permettent de réduire la complexité de la programmation du matériel. Alors que de nombreux langages de programmation existent, ils ont été conçus pour exploiter deux grands types d'architectures principalement. Ces deux types d'architectures fondamentales sont l'architecture de Von Neumann et l'architecture de Harvard (cf. Figure 3.1).

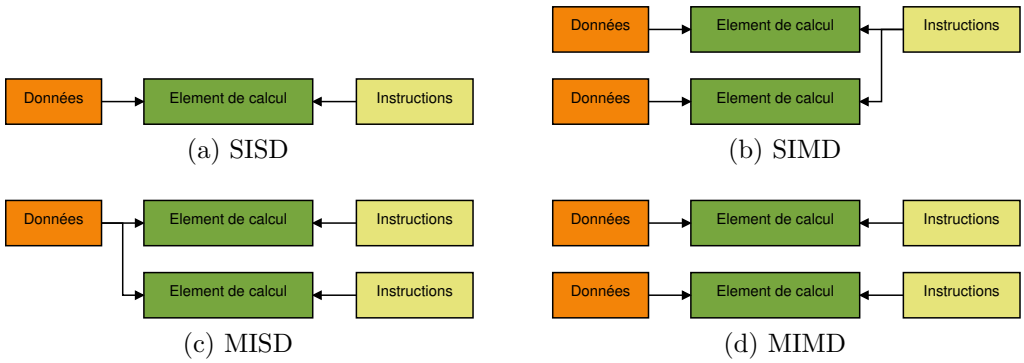


FIGURE 3.2 – Les quatre catégories de machines possibles selon la taxonomie de Flynn.

L'architecture de Von Neumann Le mathématicien hongrois John Von Neumann fut le premier à soumettre la description d'un ordinateur électronique en 1945 selon une architecture portant désormais son nom. Cette dernière est composée des quatre principaux composants précédemment énumérés. Depuis, quasiment tous les processeurs suivent ce design en l'améliorant à travers diverses modifications. Par exemple, la famille Motorola 68xxx ou la famille Intel 80x86. Les trois principales améliorations apportées au modèle de Von Neumann sont : les mémoires caches, l'exécution des instruction en pipeline et l'exécution simultanée de la même instruction sur plusieurs données (SIMD et superscalaire). En effet, au cours des années, la bande passante de la mémoire n'évoluant pas à la même vitesse que la puissance de calcul des processeurs, l'écart se creuse. Une partie de plus en plus grande des transistors est alors dédiée à masquer la latence grandissante des transferts de données entre le processeur et la mémoire. D'où l'apparition des multiples niveaux de caches (L1, L2, L3). Cependant, si cette stratégie a perduré jusque là, c'est qu'elle s'est avérée économiquement rentable. Or, pour continuer sur cette voie, le gain en performance doit être susceptible de justifier le coût de développement/production.

L'architecture de Harvard L'architecture de Harvard, du nom de l'université où cette architecture fut mise en place pour la première fois avec le Mark I en 1944, se distingue de l'architecture de Von Neumann par la séparation des données et des instructions dans deux espaces mémoires distincts. Ce modèle peut se montrer plus rapide à technologie équivalente que celui de Von Neumann. Cependant, le gain en performance s'obtient au prix d'une complexité accrue de l'architecture. Cette dernière est surtout utilisée dans des microprocesseurs spécialisés pour des applications temps réels comme les *Digital Signal Processor* (DSP).

De nombreuses architectures ont depuis été développées multipliant le nombre d'unité de contrôle ou de mémoire de l'architecture de Von Neumann. La taxonomie de Flynn, proposée par Michael J. Flynn en 1966, est une classification divisant les ordinateurs en quatre catégories. Le Tableau 3.1 résume la classification de ces architectures en fonction de la répartition des flux de contrôle et de données (Figure 3.2).

Peut traiter :	une seule donnée à la fois	plusieurs données à la fois
une instruction à la fois	SISD	SIMD
plusieurs instructions à la fois	MISD	MIMD

TABLE 3.1 – Taxonomie de Flynn.

Single Instruction stream, Single Data stream (SISD) Ce sont les premiers ordinateurs. Ils disposaient d'une architecture séquentielle incapable d'exprimer la moindre forme de parallélisme. Cette catégorie correspond à l'architecture de Von Neumann.

Single Instruction stream, Multiple Data streams (SIMD) Il s'agit d'une architecture qui commence à exprimer un certain parallélisme : le parallélisme de données. La même instruction est exécutée sur plusieurs données simultanément. De nos jours, la majorité des ordinateurs possèdent des jeux d'instructions étendus (MMX, SSE, AVX, AltiVec, Neon) capables d'effectuer des calculs sur plusieurs données différentes à la fois, ce qui les classe d'office dans la catégorie SIMD. Les processeurs vectoriels (machines Cray) sont un type particulier de SIMD.

Multiple Instruction streams, Single Data stream (MISD) Il s'agit d'une architecture dans laquelle une même donnée est traitée par plusieurs processeurs en parallèles. Assez rare, parmi elles nous pouvons quand même citer les architectures systoliques et cellulaires. Cette catégorie peut être utilisée dans le filtrage numérique, la cryptographie pour les tests de robustesse et la vérification de redondance dans les systèmes critiques.

Multiple Instruction streams, Multiple Data streams (MIMD) Ce sont des architectures qui peuvent exécuter des instructions différentes sur des données différentes. Les processeurs multi-cœurs, les multi-processeurs et les clusters de PC font partie de la catégorie MIMD. Cette catégorie peut être décomposée en deux sous-catégories : le *Single Program, Multiple Data* (SPMD) et le *Multiple Programs, Multiple Data* (MPMD). On pourrait facilement confondre le SPMD avec le SIMD. La différence est que le SIMD limite le parallélisme à l'exécution d'une instruction sur plusieurs données simultanément tandis qu'avec le SPMD, les unités de calcul peuvent effectuées des instructions différentes, seulement elles appartiennent au même programme.

3.2 Les accélérateurs

En 1965, Gordon E. Moore, cofondateur de la société Intel, prédit que le nombre de transistors sur une puce de silicium doublera tous les 18 mois. En 1975, il ajustera finalement cette valeur à tous les 2 ans [79] (Figure 3.3). Bien qu'il ne s'agisse pas d'une loi physique mais juste d'une extrapolation empirique, cette prédiction s'est avérée étonnement vraie depuis son énoncée. Les performances des processeurs étant directement liées, entre autres, au nombre de transistors, la course effrénée à la miniaturisation des composants par les fabricants de processeurs était lancée.

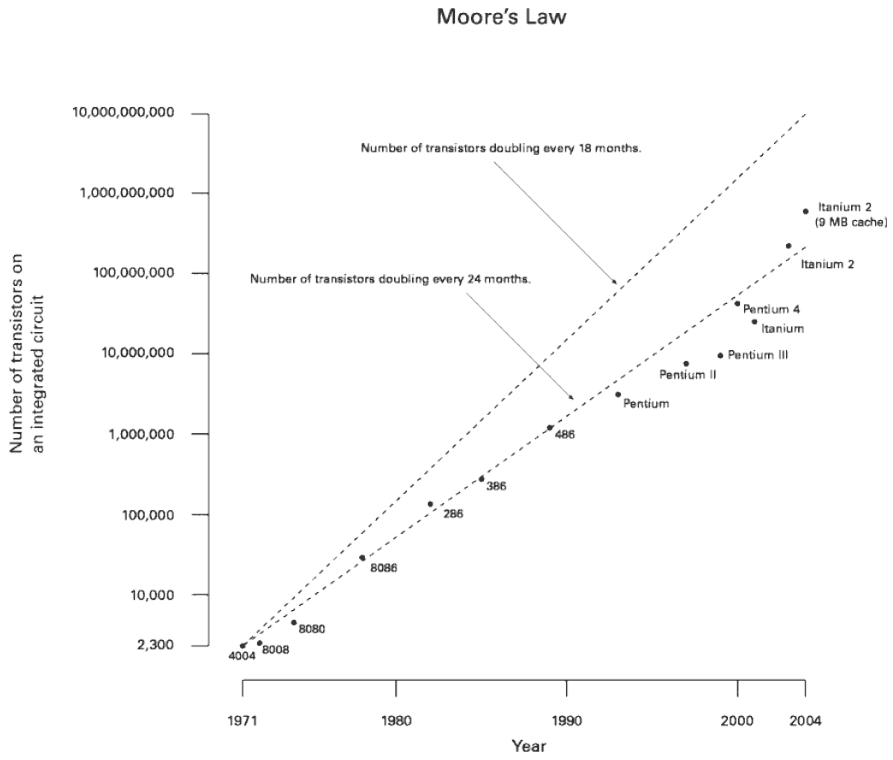


FIGURE 3.3 – La loi de Moore : la densité des transistors sur une puce de silicium doublera tous les deux ans.

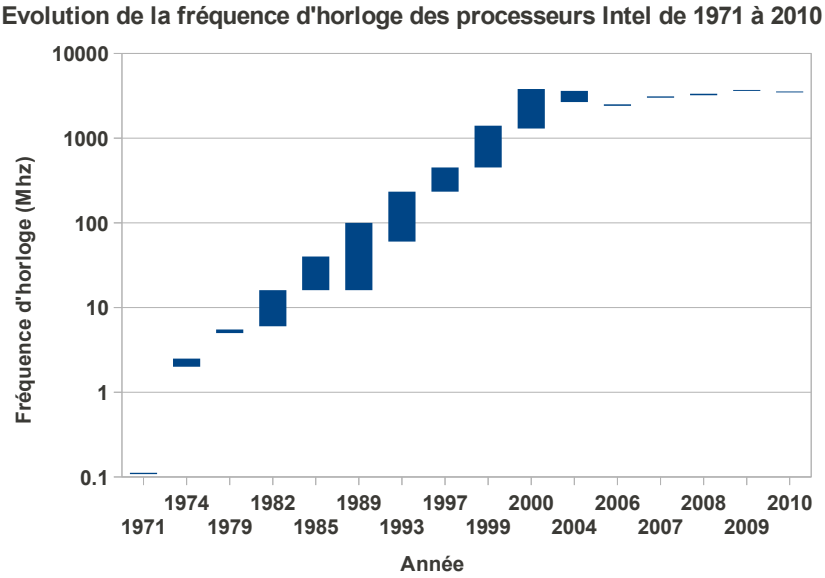


FIGURE 3.4 – Escalade de la fréquence d'horloge des processeurs jusqu'au début des années 2000 puis stabilisation de la fréquence autour de 3-4 GHz. Donnée obtenues sur le site d'Intel [1].

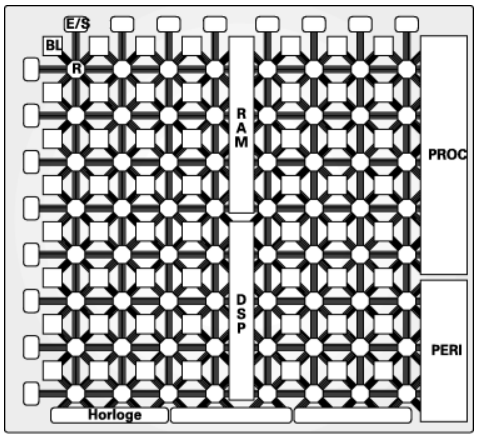
Depuis les années 1970 et jusqu'en 2000, durant la *free lunch era* [114], l'augmentation exponentielle de la cadence des processeurs (cf. Figure 3.4) poussait dans la même direction les performances des programmes séquentiels. Il suffisait alors d'acheter la dernière génération de processeurs pour augmenter les performances de son programme séquentiel. Cela était sans compter sur le mur physique qui se dressait devant eux. En effet, pour fonctionner un processeur doit modifier l'état de transistors. Or, la puissance électrique consommée pour une telle modification, mieux connue sous la terminologie puissance dynamique $P_{dynamique}$, est exprimée de la sorte :

$$P_{dynamique} = \frac{1}{2} C \times V^2 \times f$$

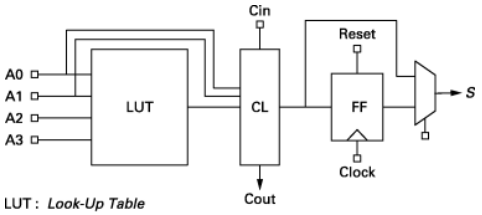
Avec C la charge capacitive, V le voltage opérationnel et f la fréquence de fonctionnement.

Depuis 2004, avec la consommation électrique grandissante et la montée en flèche de la fréquence d'horloge, les systèmes de dissipation thermique mis en place ne suffisent plus à assurer une température de fonctionnement adéquate. La densité de puissance développée par les microprocesseurs excède alors celle des plaques chauffantes. C'est ainsi que l'architecture NetBurst a progressivement été abandonnée à partir de 2006 empêchant Intel d'atteindre la fréquence prévue à l'origine pour le Pentium 4 (entre 7 et 10 GHz). La fréquence d'horloge des processeurs est désormais stabilisée aux alentours de 2-3 GHz et le paramètre en jeu est désormais le nombre de cœurs.

Dans le domaine du calcul haute performance (HPC pour High Performance Computing), les processeurs les plus performants à l'heure actuelle sont des multi-processeurs multi-cœurs :



(a) Vue générique d'un FPGA



(b) Vue générique d'un bloc logique (BL) de FPGA

FIGURE 3.5 – Architecture d'un FPGA.

Intel Xeon Sandy Bridge, AMD Opteron Piledriver Abu Dhabi, IBM POWER7 et Sun Niagara 2 (UltraSPARC T2+). D'autres solutions massivement parallèles se démocratisent aussi dans le domaine du HPC. Elles embarquent généralement ce qu'on appelle des "accélérateurs" : ce sont des architectures parallèles capable de collaborer avec le GPP pour accélérer ses calculs. Parmi les plus répandus, nous pouvons compter : le *Field-Programmable Gate Array* (FPGA), le *Cell Boradband Engine* (Cell BE) et le GPU. Le FPGA a depuis longtemps été un outil de développement privilégié pour la conception de circuits logiques. Le *Cell Boradband Engine Architecture* comme son nom l'indique a été conçu pour une palette d'utilisation variée dans un contexte où les architectures parallèles prenaient leurs marques. Le GPU a de son côté évolué pour adapter son pipeline graphique en une architecture massivement parallèle conciliant graphisme et Calcul Haute Performance. A.R. Brodtkorb et al. [19] présentent les travaux existants sur ces trois architectures.

3.2.1 Le Field Programmable Gate Array (FPGA)

Le FPGA est un circuit intégré à portes logiques (re)programmables qui excelle dans les opérations de logique combinatoire. Une fois programmé, il se comporte comme une architecture dédiée à une application spécifique (ASIC pour Application Specific Integrated Circuit). Afin de tirer le maximum de bénéfice d'un FPGA, il est d'usage de cacher la latence grâce à une utilisation pipeline favorisant ainsi le débit.

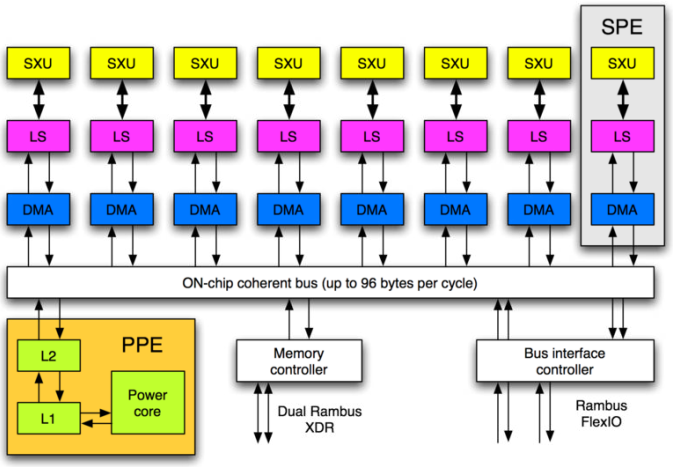


FIGURE 3.6 – Architecture du processeur Cell.

Les deux principaux composants d'un FPGA sont les *blocs logiques* et les *blocs de routage* (cf. Figure 3.5). Un bloc logique est de manière générale constitué d'une table de correspondance (LUT pour Look-Up-Table) et d'une bascule (Flip-Flop). La LUT sert à implémenter des équations logiques ayant généralement 4 à 6 entrées et une sortie. Elle peut toutefois être considérée comme une petite mémoire, un multiplexeur ou un registre à décalage. Le registre permet de mémoriser un état (machine séquentielle) ou de synchroniser un signal (pipeline). Les blocs logiques, présents en grand nombre sur la puce (de quelques milliers à quelques millions) sont connectés entre eux par une matrice de routage configurable. Ceci permet la reconfiguration à volonté du composant, mais occupe une place importante sur le silicium et justifie le coût élevé des composants FPGA. La topologie est dite « Manhattan », en référence aux rues à angle droit de ce quartier de New York.

Sa programmation en VHDL ou Verilog est assez ardue et nécessite une bonne connaissance du matériel. Le temps de développement est également non négligeable avec des temps de l'ordre du mois pour du prototypage et de l'année pour les optimisations. Certaines solutions de développement alternatives comme Mitrion-C, Viva et Catapult C facilitent sa programmation. Viva, un processus de codage graphique à la façon de LabView, a été utilisé par la NASA avec grand succès.

3.2.2 Le Cell Broadband Engine (Cell BE)

Le Cell BE est une architecture hétérogène développée par IBM, Sony et Toshiba [6]. Il dispose d'un processeur GPP traditionnel : le *Power Processing Element* (PPE), un PowerPC 64 bits légèrement modifié et de 8 co-processeurs : les *Synergystic Processing Elements* (SPE), des processeurs vectoriels SIMD, qui traitent le travail fournit par le PPE. Le PPE et les SPE sont reliés entre eux par un bus en anneaux : l'*Element Interconnect Bus* (EIB), disposant

d'une très grande bande passante. Le PPE détient des caches de niveau L1 (64 Ko) et L2 (512 Mo) tandis que chaque SPE inclus une mémoire locale (LS pour Local Storage) de 256 Ko de type SRAM haute vitesse (cf. Figure 3.6). Le Cell est intéressant d'un point de vue consommation énergétique. Par contre, sa programmation est complexe. Il existe actuellement deux compilateurs pour Cell : GCC et IBM XL. Afin de faciliter sa programmation, d'autres langages élevant le niveau d'abstraction matériel ont été développés. Les quatre principaux sont : OpenMP, MPI, CellSs [14] et Sequoia [37]. Parmi les travaux plus récents, nous pouvons citer : Skell BE [100] et Cell-MPI.

Pour information, le PowerXCell 8i, une variante du processur Cell BE, est utilisé dans le premier supercalculateur à petaflops *Roadrunner*, listé ordinateur le plus rapide au monde en juin 2009. Il faisait également partie des supercalculateurs les plus "verts" du top500 de novembre 2009 [1].

Il utilise une technologie de gravure 65 nm afin de réduire la consommation électrique de la précédente version (90 nm) tout en maintenant une fréquence de fonctionnement à 3.2 GHz. Le nombre d'unités de calcul flottant double précision a été augmenté ; les SPE délivrent désormais jusqu'à 190 TFlops, soit près de 5 fois plus qu'originellement. La mémoire Rambus XDR est remplacée par de la DDR2 et sa capacité mémoire peut atteindre les 32 Go.

3.2.3 Le Graphics Processing Unit (GPU)

Initialement conçu comme une carte annexe pour décharger le GPP des tâches de rendu graphique, la carte graphique est devenue un composant complexe, très spécialisée, quasiment imbattable dans sa catégorie. Rapidement, son utilité s'est imposée et on l'embarque désormais dans quasiment tous les ordinateurs. Avec plusieurs centaines de processeurs scalaires et une bande passante mémoire imposante, le GPU dispose d'une puissance de calcul théorique dépassant largement celle des GPP.

Au début du XXIè siècle, plusieurs chercheurs-pionniers attirés par cette puissance de calcul phénoménale pour quelques centaines d'euros se sont lancés dans leur utilisation pour du calcul scientifique parallèle. Bien entendu, il fallait pour cela être suffisamment familiarisé avec le fonctionnement du pipeline graphique afin de traduire l'application à porter sur GPU dans un langage de programmation dédié aux traitements graphiques : OpenGL, HLSL ou Cg. L'exploitation de ces cartes graphiques était donc assez fastidieuse à cause des contraintes de programmation adaptées à un environnement de développement purement graphique.

Depuis 2007, Nvidia a transformé l'architecture de ses cartes graphiques, unifiant les shaders de son pipeline graphique. Ainsi est né *Compute Unified Device Architecture* (CUDA), une architecture pilotée par des API bas et haut niveau programmable en C ou en Fortran avec des extensions de langage. A travers une politique marketing agressive, des événements tels que la

1. http ://www.green500.org/lists/green200911

célèbre GPU Technology Conference (GTC), des accords entre les universités et les entreprises, des centres d'excellence pour l'apprentissage des méthodes de programmation GPGPU, Nvidia a entrepris et réussi la démocratisation du GPGPU. Pour faciliter le portage, Nvidia fourni des outils pour accompagner l'utilisateur dans la programmation de ses GPU : *cuda-gdb*, *profiler*, *occupancy calculator*. NSIGHT, le dernier débogueur de Nvidia, est relativement complet. Il permet de déboguer dans le détail, permettant au programmeur d'aller jusqu'à vérifier le contenu du registre d'un thread spécifique du programme lancé. Plusieurs librairies évitant de réinventer la roue sont également à la disponibles aux programmeurs :

- NVIDIA Performance Primitives [2] (NPP), l'équivalent d'Intel Performance Primitives (IPP)
- AccelerEyes [3] Jacket est une toolbox pour accélérer les calculs MATLAB sur GPU tandis que AccelerEyes ArrayFire est une bibliothèque incluant plusieurs fonctions de mathématiques, traitement du signal et de l'image, statistiques, etc
- GpuCV [92], l'équivalent d'Open source Computer Vision (OpenCV)
- *Matrix Algebra on GPU and Multicore Architectures* (MAGMA) [4] est une bibliothèque d'*Algèbre Linéaire Dense* (DLA pour Dence Linear Algebra) similaire à *Linear Algebra PACKage* (LAPACK) mais adaptée aux architectures hétérogènes GPP-GPU
- cuFFT [5] pour le calcul des *Transformées de Fourier Rapide* (FFT pour Fast Fourier Transform)
- cuBLAS [6] la version CUDA de *Basic Linear Algebra Subprograms* (BLAS).

Le paradigme de programmation CUDA sera étudié plus en profondeur dans la section 3.5.

3.3 Le GPGPU appliqué au traitement de l'image

Le traitement de l'image est une discipline de l'informatique et des mathématiques appliquées qui étudie les images numériques dans le but de transformer (améliorer une photographie avec un logiciel comme PhotoShop ou GIMP, par exemple), extraire des informations (accompagné le médecine dans un diagnostic, par exemple) ou bien interpréter des informations (alerter dans le cas d'intrusions par vidéosurveillance, par exemple).

Le besoin d'automatiser le traitement des images est rapidement devenu une évidence. « Dès 1960, il fallait analyser 10 000 à 100 000 images par expérience dans les chambres à bulles pour déterminer les trajectoires de milliers de particules grâce à plusieurs caméras réparties » rappelle Serge Castan. Cependant, plusieurs facteurs technologiques limitaient l'utilisation de l'informatique dans ce domaine. Puis des images de meilleure qualité, présentant moins d'artefacts et plus riches en informations, une capacité de stockage suffisante pour contenir les volumes de données à traiter et des processeurs offrant un temps de traitement relativement

2. http ://developer.nvidia.com/cuda/nvidia-performance-primitives
3. http ://www.accelereyes.com/
4. http ://icl.cs.utk.edu/magma/
5. http ://developer.nvidia.com/cuda/cufft
6. http ://developer.nvidia.com/cuda/cublas

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011 IBM	1572864	16324.75	20132.66	7890.0
2	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	705024	10510.00	11280.38	12659.9
3	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	786432	8162.38	10066.33	3945.0
4	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012 IBM	147456	2897.00	3185.05	3422.7
5	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010 NUDT	186368	2566.00	4701.00	4040.0
6	DOE/SC/Oak Ridge National Laboratory United States	**Jaguar** - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009 Cray Inc.	298592	1941.00	2627.61	5142.0
7	CINECA Italy	**Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	163840	1725.49	2097.15	821.9
8	Forschungszentrum Juelich (FZJ) Germany	**JuQUEEN** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	131072	1380.39	1677.72	657.5
9	CEA/TGCC-GENCI France	**Curie thin nodes** - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012 Bull	77184	1359.00	1667.17	2251.0
10	National Supercomputing Centre in Shenzhen (NSCS) China	**Nebulae** - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010 Dawning	120640	1271.00	2984.30	2580.0

FIGURE 3.7 – Liste des 10 supercalculateurs les plus puissants au monde en juin 2012. Encerclés en rouge sont les supercalculateurs qui embarquent des GPU. Les colonnes R_max et R_peak correspondent respectivement au performances maximales théorique et aux performances crêtes atteintes. L'efficacité d'utilisation des supercalculateurs, de l'ordre de 80 à 90% sans GPU, chute aux alentours de 50 à 60% quand les GPU sont utilisés comme accélérateurs.

court ont fait du traitement de l'image par ordinateur un sujet d'actualité.

L'explosion de la puissance de calcul mise à disposition des utilisateurs permet désormais des opérations mathématiques longues et complexes. De plus, le domaine de la vision par ordinateur est en plein essor grâce aux possibilités offertes par la combinaison du traitement de l'image et de l'intelligence artificielle (réseaux de neurones, machines à vecteurs de support, algorithmes de génétique, etc). L'application du traitement de l'image dans le domaine du médical présente aussi une forte croissance : assistance au chirurgien par un traitement in situ des images médicales, interprétation des Images à Résonances Magnétiques (IRM) afin de mieux comprendre le fonctionnement du cerveau, etc.

On entend beaucoup parler du GPU pour l'accélération des calculs de traitement de l'image de nos jours. Ses nombreux avantages en font l'un des moyens de calcul favoris pour accélérer le traitement d'images :

- son accessibilité : il est présent dans tous les ordinateurs actuels et ne nécessite donc aucuns frais et/ou installation de matériel supplémentaires.
- sa simplicité de programmation : il ne nécessite pas de reprogrammation du hardware (FPGA) et la gestion des canaux de communication est moins fastidieuse que sur le Cell.
- la scalabilité de ses performances : si une cartes d'entrée de gamme n'est pas suffisamment performantes, il est possible d'acquérir une carte plus puissante à faible coût d'acquisition ou d'attendre la prochaine génération de cartes pour espérer un gain supplémentaire en performances (à condition qu'il n'y ait pas de modifications majeurs dans l'architecture des nouvelles cartes). Nous voyons ici se transposer le comportement de la *free lunch era* des processeurs GPP aux cartes GPU.
- son modèle de programmation : il dispose de plusieurs centaines de cœurs organisés en groupes de façon non-hiérarchique et gère plusieurs milliers de threads en parallèles.

Les deux principaux inconvénients que l'on peut reprocher au GPGPU sont sa consommation électrique et une exploitation optimale (proche des performances crêtes théoriques) assez difficile à atteindre. Pour l'anecdote, le TOP500[7] répertorie la liste des 500 supercalculateurs les plus puissants au monde. La liste est mise à jour deux fois par an : en juin et en novembre. Deux indices sont donnés : le premier représente la performance maximale atteinte avec le benchmark LINPACK [33] et le second est la performance crête théorique. La Figure 3.7 est une capture d'écran du classement retrouvé sur leur site web[8]. On observe que l'efficacité d'utilisation des supercalculateurs est de l'ordre de 80 à 90% sans usage des GPU et elle chute aux alentours de 40 à 60% quand des GPU sont utilisés comme accélérateurs. Sans doute pourrions-nous avancer que l'exploitation des GPU n'est pas identique à celle des GPP. Cependant, le supercalculateur Roadrunner composé de nombreux processeurs Cell montre une efficacité d'utilisation de l'ordre de 75%.

Du coup, en quoi cette nouvelle technologie est-elle révolutionnaire sachant que le portage d'algorithmes sur GPU nécessite quand même une certaine démarche d'Adéquation Algorithme Architecture et une connaissance approfondie de son architecture. Si on optimisait le GPP

7. http ://www.top500.org
8. http ://www.top500.org

avec des techniques d'optimisation aussi poussées que sur GPU serait-il possible d'obtenir des performances tout aussi importantes voire même meilleures ? Serions-nous victimes de la séduction opérée par la promesse alléchante de facteurs de gains démentiels ? Accorder autant de crédibilité au GPGPU parce que c'est une architecture massivement parallèle avec plusieurs centaines de cœurs comparée aux quelques cœurs d'un GPP dernière génération n'est-il pas un raccourci quelque peu rapide ?

Victor W. Lee et al. [68] se sont penchés sur le présumé mythe des gains pouvant atteindre un facteur x1000. Cela est d'autant plus intriguant que les gains présentés dans les articles balayent une fourchette plutôt large allant de x10 à x1000. Leur conclusion est que les chiffres avancés ne prennent pas en considération l'écart des efforts d'optimisation entrepris entre une solution GPP et une solution GPU. On ne peut pas comparer une implémentation GPU optimisée à une implémentation GPP mono-thread. Certains articles comparent les temps d'exécution d'une solution GPU sans considérer les temps de transfert GPP $\leftrightarrow$ GPU. Ces résultats sont donc biaisés, attribuant au GPU davantage de crédit qu'il n'en mérite. Victor W. Lee et al. [68] retiennent toutefois la suprématie des GPU avec un gain en performance de l'ordre de x2.5 en moyenne pour les différentes implémentations qu'ils ont pu étudier.

Les gains apportés par le GPU sont donc bien réels mais ne sont pas aussi facilement atteignable que l'effervescence autour du GPU pourrait le faire croire. Plusieurs facteurs en dépendent dont l'application, la taille du problème à résoudre, la fréquence des échanges entre GPP et GPU, les choix d'optimisation, etc

Dans le domaine de l'imagerie médicale, Gac [42] a entrepris des optimisations poussées sur diverses architectures pour l'implémentation d'un algorithme de reconstruction 3D à partir d'images médicales acquises par tomographie. Il présente des gains au moins 5x supérieurs sur GPU par rapport au GPP et au FPGA. Smelyanskiy et al.

[108] obtiennent des performances 5.8x meilleurs sur un Intel Nehalem Quad-Core comparé à une implémentation scalaire optimisée tournant sur un Harpertown simple cœur. Parallèlement, une version optimisée pour Nvidia GTX280 atteint des performances de 5 à 8x supérieures à l'implémentation sur un Harpertown. Encore une fois, nous observons des performances légèrement supérieures apportées par l'utilisation du GPU.

Dans le domaine des mathématiques appliquées, Volkov et al. [126] a obtenu des gains de 2x à 4x supérieurs sur 8800GTX et GTX280 à ceux obtenus sur un Core2Quad pour des optimisations poussées d'opérations d'algèbre linéaire dense.

Satish et al. [102] décrivent des algorithmes de tri parallèles hautement efficaces avec des performances 23% plus rapide, en moyenne, sur GPU qu'une implémentation GPP très soigneusement optimisée.

Castaño-Dìez et al. [23] rapportent des gains de 10 à 20x supérieurs comparés aux résultats sur des processeurs conventionnels pour l'implémentation d'algorithmes de transformées

spatiales, d'opérations de Fourier et de reconnaissance de formes.

Dans le domaine du contrôle non destructif, Pedron et al. [91] comparent les temps d'exécution d'un algorithme de reconstruction de trajectoires par ultrasons sur GPP et GPU. Les deux implémentations soigneusement optimisées apportent des gains entre 15 et 60x supérieurs sur GPP et entre 69 et 548x supérieurs sur GPU pour différents jeux de données. Au final, le GPU apporte un gain 4 à 10x supérieur par rapport au GPP.

Dans le domaine de la vision par ordinateur, Temizel et al. [115] étudient les performances d'algorithmes de soustraction d'arrière-plan et de corrélation appliquées au domaine de la vision par ordinateur et plus particulièrement à la vidéosurveillance. Leur analyse se concentre sur une comparaison entre CUDA et OpenCL, concluant que CUDA apporte de meilleures performances qu'OpenCL. Néanmoins, ils comparent également leur implémentation GPU avec une implémentation GPP. Ils obtiennent un gain environ 4x supérieurs avec une implémentation OpenMP sur un quad-cœur Intel i7 920 et des performances 11.6 à 89.8x supérieurs sur GPU.

D'autres gains intéressants ont été révélés dans le domaine de la vision par ordinateur [113, 99, 40, 92, 62, 64], de l'imagerie médicale [65, 106, 39, 57, 111, 104], de la science de Terre [127, 77], de la science des matériaux [42, 55] et leur utilisation s'est étendue et diversifiée à tous les domaines de l'ingénierie nécessitant des calculs de simulation importants.

Cependant, bien que le GPGPU soit très attractif, sa programmation nécessite un apprentissage et une maîtrise de ses espaces mémoires. Afin d'étudier au mieux les goulets d'étranglements lors du portage d'un algorithme sur GPU, rien ne vaut la pratique. Nous tâcherons donc, dans le Chapitre 4, de présenter l'implémentation d'un algorithme simple de traitement d'image. Ce dernier nous fournira matière à évaluer les contraintes d'implémentation d'un algorithme sur GPU.

3.4 Les langages de programmation pour le GPGPU

Alors que Nvidia détenait son langage de programmation propriétaire pour GPGPU : CUDA, son concurrent principal *Advanced Micro Devices* (AMD), constructeur de processeurs et de carte graphiques, suit la tendance avec son architecture *Close To Metal* (CTM) et son propre environnement de développement *Compute Abstraction Layer* (CAL) (bas niveau) et Brook+ (haut niveau). Pour la programmation haut-niveau, AMD a adopté une version étendue de Brook, langage originellement développé à Standford [21]. Cette version étendue d'ATI est nommée Brook+ [8]. Cependant, le niveau d'abstraction matérielle semble être plus élevé que CUDA et permettrait moins d'optimisations matérielles. Un manque de performances préjudiciable auquel CAL aurait pu remédier par son approche bas-niveau si sa prise en main n'avait pas été aussi difficile.

Avec la démocratisation des architectures hétérogènes, plusieurs travaux visent à établir un langage de programmation universel. Alors que Stratton et al. [112] décrivent certaines techniques pour l'implémentation efficace du modèle de programmation CUDA sur des processeurs GPP multi-cœurs, ils suggèrent aussi son adaptation pour la spécification d'un modèle de programmation universel dont le code parallèle serait portable et accessible sur une panoplie d'architectures parallèles. Stratton et al. espèrent également voir l'écart de performances entre du code C optimisé et son équivalent CUDA sur GPP se réduire au fur et à mesure que ce langage multi-plateforme mûrisse.

Apparemment, d'autres personnes pensaient pareil. OpenCL est un standard récent, ratifié par le groupe Khronos, pour programmer des ordinateurs hétérogènes. Khronos est majoritairement constitué de compagnies dont les centres d'intérêt sont le calcul parallèle, le graphisme, les mobiles, le divertissement et l'industrie du multimédia. Les standards qu'ils développent accessibles à tous sont à but lucratif : grâce à des standards comme OpenGL, ils créent des opportunités commerciales sur le marché. Les acteurs du hardware majeurs et plusieurs acteurs parmi les vendeurs de software font partie du bureau de ratification et ont bon espoir qu'OpenCL vivent le même succès qu'OpenMP. Khronos a commencé à travailler sur OpenCL en juin 2008 suite à une proposition d'Apple. Le bureau de ratification a reçu la proposition de standard en octobre et l'a ratifié en décembre 2008.

OpenCL est constitué d'un langage de programmation adapté aux accélérateurs et d'un API pour orchestrer l'appel aux kernels OpenCL sur les accélérateurs à partir de la plateforme. Le langage de programmation est basé sur le C99, avec une philosophie et une syntaxe similaire à CUDA exploitant le modèle *Single Program Multiple Data* (SPMD) pour la programmation parallèle. Le parallélisme de tâches est également supporté par le lancement de tâches multiples sous forme de kernels mono-thread, exprimant le parallélisme sur des vecteurs de données de largeur 2-16. Le choix d'exécuter les kernels en pile de commandes permet une exécution des tâches dans le désordre tout en exprimant le parallélisme de tâches à fine granularité. Les tâches exécutées dans le désordre sont synchronisées par barrières ou explicitement en spécifiant les dépendances. La synchronisation entre piles de commandes réparties sur différents accélérateurs est aussi explicite. Le standard défini une charte d'utilisation pour une utilisation conforme d'OpenCL, comme il a été le cas pour OpenGL. Des extensions optionnelles sont également définies, incluant les fonctions double précision et les fonctions atomiques. Apple a déjà incorporé OpenCL dans Mac OS X Snow Leopard et les deux constructeurs de cartes graphiques NVidia et AMD ont mis sur le marché des compilateurs beta. Il est également possible que des compilateurs pour processeurs Cell apparaissent, puisque l'équipe Cell BE de chez IBM a participé au standard. L'existence d'un support pour les FPGA commence à voir le jour [107].

A cause de sa souplesse de programmation multi-architecture, OpenCL reste généralement moins efficace que le langage de programmation CUDA dédié aux GPUs NVidia. Karimi et al. [60] et Temize et al. [115] démontrent les meilleurs résultats obtenus avec CUDA comparés à OpenCL. Danalis et al. [29] proposent un banc de test permettant de comparer les performances de CUDA et d'OpenCL. Il en ressort que bien qu'OpenCL puisse avoir des performances proches de CUDA, ces dernières chutes très rapidement quand les kernels implémentés sont

non triviaux (FFT, Dynamique Moléculaires par exemple).

Depuis la sortie du standard OpenCL, AMD l'a adopté comme langage de programmation pour ses GPUs ; les solutions précédentes sont donc dépréciées.

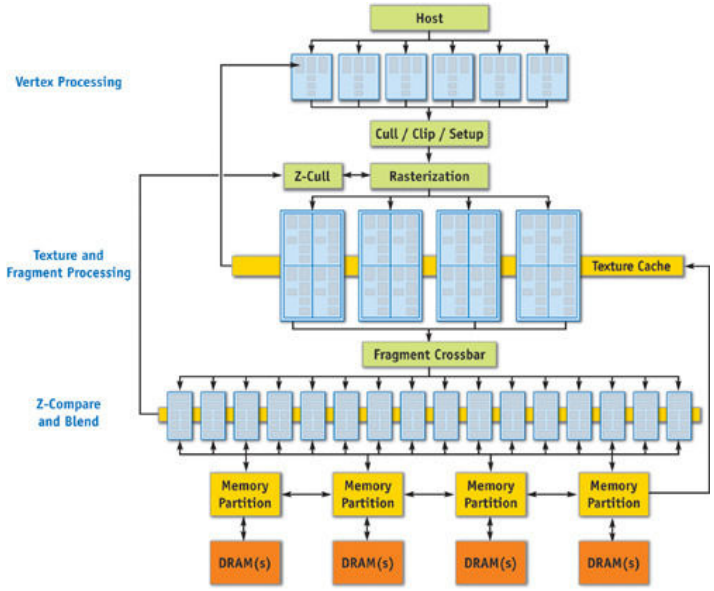
3.5 Le paradigme de programmation CUDA

CUDA étant une plateforme matérielle et logicielle, nous distinguerons l'architecture et le paradigme de programmation. Puis nous discuterons des principaux goulots d'étranglement dans la programmation GPGPU.

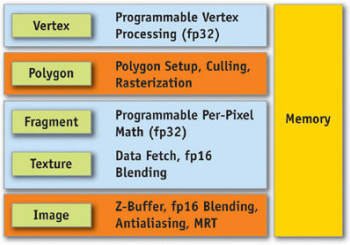
3.5.1 L'architecture

Comme son nom l'indique, *Compute Unified Device Architecture* représente une révolution du modèle architectural des cartes graphiques NVidia. Les cartes graphiques NV20 (2001, GeForce 3 family) et NV40 (2004, GeForce 6800 family) étaient constituées d'un pipeline graphique composé de trois principales étapes : le *vertex shader*, le *rasterizer* et le *fragment shader* (Figure 3.8). Depuis la G80 (2006, GeForce 8800 family), dans la perspective de transformer le pipeline graphique en une machine parallèle, NVidia a unifié l'architecture des processeurs du *vertex shader* et du *fragment shader* (Figure 3.9). Son successeur, la GT200, sortie en juin 2008, apporte surtout des améliorations au niveau de la puissance délivrée et des ressources allouables. Peu de modifications bousculent l'architecture de la G80 mis à part le nombre de *Streaming Multiprocessor* (SM) par *Thread Processing Cluster* (TPC) qui passe de 2 à 3 (Figure 3.10). Avec la GT300 (ou GF100 ou Fermi), NVidia fournis davantage de fonctionnalités : une augmentation du nombre de processeurs de calcul flottant à double précision, un développement de la hiérarchie des caches entre la mémoire globale et les processeurs scalaires, une protection ECC contre les erreurs éventuelles pendant les transactions de données, une unification de l'espace d'adressage des espaces mémoires de la carte et du GPP permettant d'exécuter du code C++ directement sur le GPU et la possibilité d'exécuter 16 kernels simultanément (limité à un auparavant). Avec l'architecture Fermi, les GPU NVidia sont devenus des architectures parallèles dédiées au calcul scientifique à part entière (Figure 3.11). Depuis, une dernière génération de cartes a fait irruption, la Kepler, améliorant la souplesse d'utilisation des GPU dans le calcul parallèle sur cluster multi-GPU.

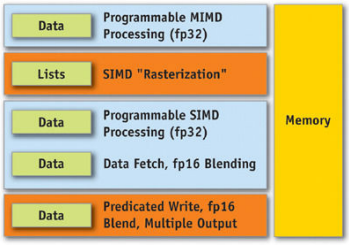
Bien que ces différentes générations de cartes soient toutes compatibles avec le paradigme de programmation CUDA, leur évolution est liée à des modifications architecturales qui se répercutent dans certaines mesures sur le modèle de programmation. Afin de distinguer les fonctionnalités CUDA accessibles par les différentes générations de carte, le terme *compute capability* [86] est employé par le constructeur. Parmi ces fonctionnalités nous pouvons lister : les opérations atomiques, un espace d'adressage mémoire unifié, des multiplications d'entiers



(a) Diagramme de blocs représentant l'architecture.



(b) Point de vue de l'architecture lors d'un usage graphique.



(c) Point de vue de l'architecture lors d'un usage non-graphique.

FIGURE 3.8 – Différentes représentations de l'architecture des GPU GeForce 6 series avant l'unification des shaders (sources GPU Gems 2).

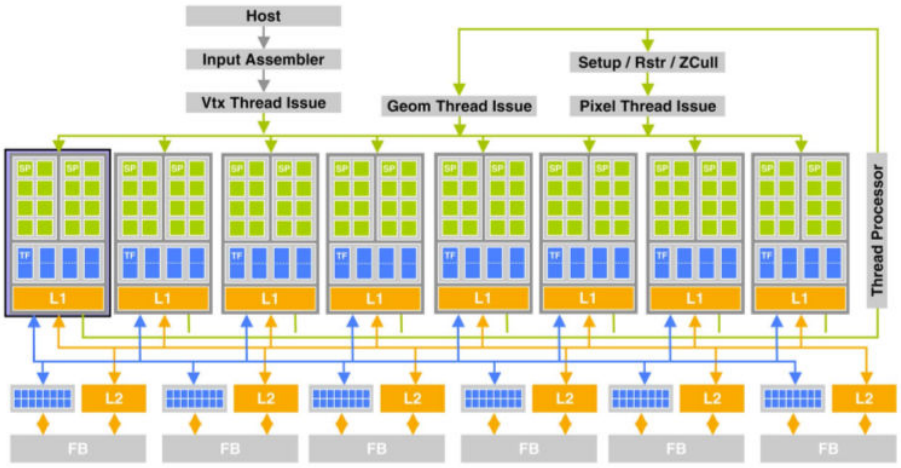


FIGURE 3.9 – La génération G80, la première architecture CUDA.

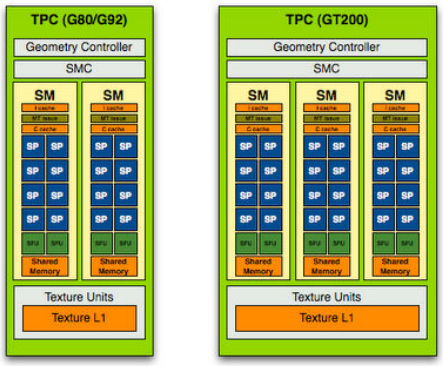


FIGURE 3.10 – Modification du nombre de multiprocesseurs (SM) par TPC entre la G80 et la GT200.

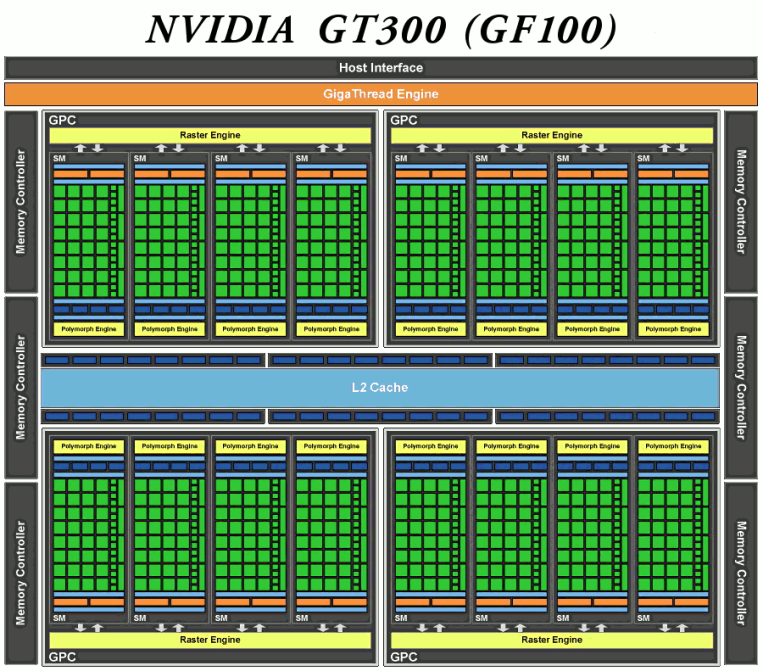


FIGURE 3.11 – L'architecture Fermi, le premier GPU réfléchi comme une architecture parallèle.

Architecture	G80	GT200	Fermi	Kepler
Compute capability	1.0 ou 1.1	1.2 ou 1.3	2.0 ou 2.1	3.0 ou 3.5

TABLE 3.2 – Compute capability pour les différentes architectures CUDA existantes.

sur 32 bits, etc. Le Tableau 3.2 présente les *compute capability* en fonction des générations de GPUs.

3.5.2 Le modèle de programmation

L'architecture la plus utilisée au cours de cette thèse étant la GT200, nous limiterons la description du paradigme de programmation à cette architecture. Ces informations restent néanmoins valables pour les architectures Fermi et Kepler malgré leurs importantes améliorations architecturales. Les paramètres tels que la quantité maximale de threads par blocs, la taille de la mémoire partagée, la hiérarchie des mémoires caches, etc peuvent toutefois changer. Afin de connaître les détails et les spécificités de chaque architecture, nous vous invitons à vous reporter à leur documentation technique sur le site de NVidia [9].

9. http ://www.nvidia.com/object/cuda_home_new.html

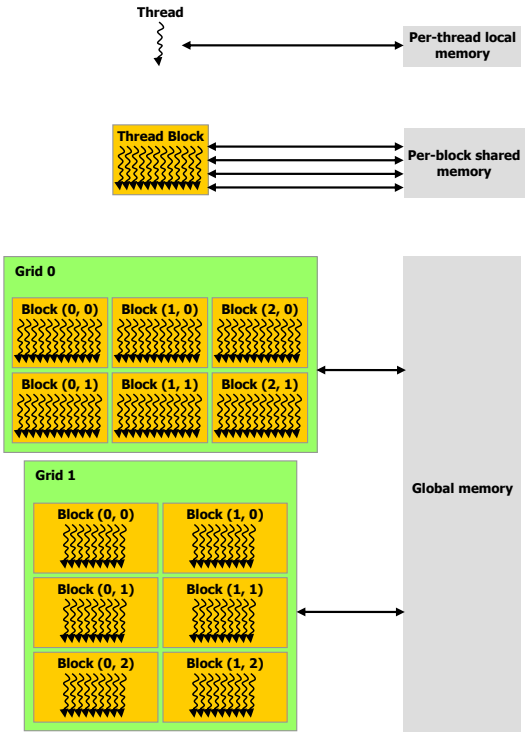


FIGURE 3.12 – Différents niveaux de granularité pour l'expression du parallèlisme.

Comme il a précédemment été présenté dans les Figures 3.9 et 3.10, les cartes graphiques sont composées de multiprocesseurs (SM pour Streaming Multiprocessor). Leur nombre peut varier d'une carte à l'autre. Chaque multiprocesseur contient 8 processeurs scalaires (CUDA cores ou SP pour Streaming Processor).

Concernant l'organisation des threads, le parallélisme est exprimé à trois niveaux de granularité (Figure 3.12) :

- l'entité la plus petite : le warp. Il regroupe 32 threads qui traitent la même instruction simultanément. Le programmeur doit donc éviter les opérations conditionnelles divergentes au sein d'un warp. Sinon, les chemins empruntés par les threads divergents sont traités successivement, allongeant les temps de calcul et brisant le modèle SIMD (qui devient donc un modèle SISD). Cette quantité est figée et si elle doit un jour être modifiée, ce sera le choix de Nvidia.

Les deux autres niveaux de granularité sont choisis par le programmeur :

- Les warps appartiennent à des blocs de threads. Chaque bloc peut contenir 512 threads maximum, sachant qu'un bloc ne peut être traité que par un seul SM mais que plusieurs blocs peuvent être traités sur le même SM. C'est le warp scheduler qui se charge de découper les threads d'un blocs en warps ; pour cela, il forme un warp en regroupant les 32 premiers threads successifs d'un bloc et ainsi de suite. A noter que le choix de la taille de bloc est une tâche sensible qui doit prendre en compte la taille maximale de celui-ci et le taux d'occupation du SM. Ce dernier point sera abordé dans le paragraphe 3.5.3.

– Les blocs de threads sont contenus dans une grille. La grille de blocs est assez grande pour contenir plusieurs milliers de blocs. Une fois que la taille d'un bloc est fixée, le choix de la taille de grille est quasi-immédiat.

3.5.3 Les goulots d'étranglement dans la programmation GPGPU

Afin d'optimiser son code sous CUDA, il est nécessaire de connaître les limites de l'architecture. Voici une liste non exhaustive des principales contraintes à prendre en considération lors de la programmation.

La bande passante du bus PCI-Express x16

Le goulot d'étranglement principal entre le GPP et le GPU est le transfert de données via PCI Express. Avec le passage du PCI Express v1.x à v2.0, le débit par ligne est passé de 250MB/s à 500MB/s. Ainsi, le port PCI-E x16 v2.0 possède une bande passante théorique maximale de 8Go/s par direction et 16Go/s en full duplex (double direction : GPP ⇒ GPU et GPU ⇒ GPP). Cependant, seules les cartes Tesla et Quaddro de génération Fermi et Kepler peuvent bénéficier du full duplex grâce à l'embarquement sur carte de deux moteurs *Direct Memory Access* (DMA). Pour les autres cartes, le débit théorique maximal est limité à 8Go/s.

L'application *bandwidthTest* fournie dans le SDK, permet d'évaluer les débits de votre configuration matérielle. Il est bon de vérifier ces derniers avant d'entamer un développement sur GPU. Ces chiffres sont les références auxquelles se tenir pour savoir combien notre implémentation proche du matériel est optimale.

Le GPU est souvent utilisé comme alternative au GPP pour des traitements lourds accompagnés de données volumineuses. Il est donc nécessaire de prendre en considération la taille des données traitées sur GPU afin d'évaluer les temps de transfert. Effectivement, les gains atteignables sur GPU peuvent être annulés par les transferts GPP ↔ GPU. Il est donc nécessaire de considérer les temps de transferts dans l'évaluation des performances du GPU pour obtenir un rendement $\frac{Transferts\ GPP \leftrightarrow GPU + Calcul\ GPU}{Calcul\ GPP}$ positif.

Une alternative pourrait être de recouvrir les transferts GPP ↔ GPU avec les calculs GPU. Pour se faire, il faut suivre les trois points suivants :
– l'espace mémoire GPP utilisé lors du transfert doit être alloué avec la fonction *cudaHostAlloc*. Grâce à cette fonction, l'espace mémoire sur le host (GPP) est alloué en *pinned memory* ; ce qui veut dire que la mémoire est allouée sur des pages mémoire fixées.
– lancer le transfert mémoire sur un stream différent du stream de lancement du kernel de calcul (les streams CUDA s'exécutent simultanément).

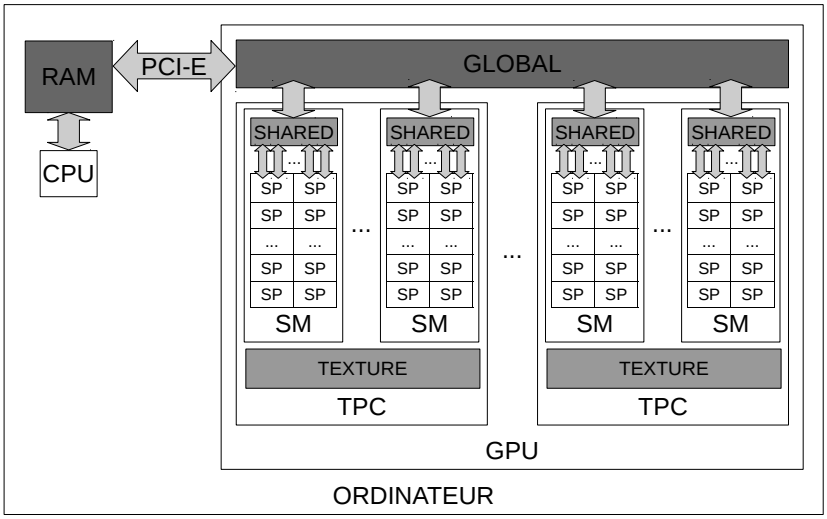


FIGURE 3.13 – Principaux espaces mémoires, gérables par l'utilisateur, mis en jeux lors de la programmation sur GPU : la RAM du GPP, la mémoire globale, la mémoire partagée et la mémoire de texture du GPU. La RAM et la mémoire globale, de couleur gris sombre, communiquent au même niveau hiérarchique ; la texture et la mémoire partagée, de couleur gris claire, appartiennent à un niveau de mémoire hiérarchique supérieur.

– utiliser les fonctions de copies mémoire asynchrones (*cudaMemcpyAsync*, *cudaMemcpy2DAsync* ou *cudaMemcpy3DAsync*)

L'organisation des espaces mémoires sur le GPU

La Figure 3.13 illustre l'organisation des espaces mémoires les plus communément utilisés sur GPU : la mémoire globale, la mémoire partagée, la mémoire de texture et les registres.

La mémoire globale est l'espace mémoire tampon entre le GPP et le GPU. Afin de communiquer avec le GPU, les données transitent inévitablement par la mémoire globale. Elle est accessible par n'importe quel multiprocesseur de la carte graphique avec des performances identiques et constantes : entre 400 à 600 cycles (très lent). Cependant, à cause de sa latence élevée, on lui préfère l'utilisation d'espaces mémoires plus proches des multiprocesseurs comme la mémoire partagée et la mémoire de texture. A noter, que la mémoire globale des générations de cartes G80 et GT200 [83, 84] ne disposent pas de niveaux de mémoire cache.

Afin de rentabiliser au maximum ces accès à la mémoire globale nous devons tâcher d'utiliser le maximum de bande passante. La bande passante est exploitée à son maximum quand les 32 threads d'un warp accèdent à un espace mémoire contigu simultanément. Cet aspect sera étudié plus en détail dans les choix d'implémentation de l'algorithme de granulométrie (4.5.2).

La texture est une mémoire cache accessible par un groupe de 2-3 multiprocesseurs (Figure 3.10) que le programmeur *bind* à la mémoire globale *i.e.* qu'il restreint la mémoire cache de texture à une région spécifique de la mémoire globale. Ses inconvénients sont sa taille limitée (∼8Ko) et le manque de cohérence des données de la mémoire de texture avec la mémoire globale après modification de cette dernière. Depuis la génération de cartes Fermi, un nouveau type de mémoire : la mémoire de surface, apporte davantage de souplesse à l'utilisation des mémoires de texture.

Plus proche des multiprocesseurs se situe la mémoire partagée : 16 Ko par multiprocesseur. La mémoire partagée d'un multiprocesseur est divisée en 16 bancs de 1Ko chacun. Cette mémoire est aussi rapide qu'un accès aux registres à condition que les accès par demi-warp (16 threads) ne se superposent pas dans le même banc. Autrement, les accès sont sérialisés.

La mémoire partagée et la mémoire de texture se distinguent principalement par le mode de gestion des données : avec la mémoire partagée, c'est le programmeur qui gère l'allocation des données et ordonne la lecture/écriture des données – la gestion des données dans la mémoire est manuelle – tandis que la mémoire de texture est une mémoire cache optimisée pour les accès à localité spatiale 2D avec des modes d'adressages et des fonctions d'interpolation câblées – la gestion des données est automatique et l'utilisateur ne gère que l'initialisation.

Ci-dessous les caractéristiques des espaces mémoires programmables (l'allocation des registres et de la mémoire locale est effectuée à l'insu de l'utilisateur).

- la mémoire globale
  - lente (400-600 cycles horloges)
  - sans mémoire cache (compute cap. < 2.0) / avec mémoire cache (compute cap. ≥ 2.0)
  - présente des contraintes d'alignement mémoire pour assurer des accès mémoire optimisés
- la mémoire partagée (shared memory)
  - rapide mais peut être ralenti par des conflits de banc mémoire
  - permet l'échange d'informations entre *Streaming Processor* (SP)
- la mémoire de texture / texture
  - une mémoire cache (6/8 Ko) optimisée pour les accès à localité spatiale 2D
  - mode d'adressage qui gère les débordements
  - fonctions de filtrage de données 1D, 2D et 3D câblées
  - il n'est pas permis d'écrire dans la mémoire de texture
- la mémoire constante
  - contenant les constantes et les paramètres fournis au kernel
  - lente mais disposant d'une mémoire cache (8 ko)
  - optmisée pour le "broadcast"

Pour résumer, il existe trois stratégies d'implémentation permettant au programmeur d'extraire les données de la mémoire globale :

- la mémoire globale (directement) : les performances seront forcément dégradées pour les générations de cartes antérieures à Fermi (les architectures G80 et GT200 ne disposant pas de niveaux de mémoire cache)
- la mémoire partagée : elle nécessite un effort d'organisation des données de la part du

programmeur. Il devra également prendre en considération l'*occupancy* pour équilibrer l'allocation mémoire partagée/registres dans son étude de performances
- la mémoire de texture : elle présente diverses fonctionnalités dont : des accès rapides pour des données à localité 2D, faciliter la gestion des débordements mémoire, proposer des calculs d'interpolations rapides, etc
- la mémoire de surface (disponible depuis la génération de cartes Fermi) : similaire à la mémoire de texture sauf qu'il est possible d'écrire dedans.

L'exploitation efficace des ressources du GPU

On désigne par taux d'occupation (ou *occupancy*) d'un multiprocesseur le pourcentage de threads créés par rapport à la quantité que peut potentiellement gérer le multiprocesseur. Ce taux dépend fortement de la répartition des ressources par multiprocesseur. Sachant que le nombre de threads par bloc est défini par le programmeur, le multiprocesseur peut ensuite traiter autant de blocs que les ressources disponibles le permettent.

Une feuille de calcul *occupancy calculator*[10], fournie par Nvidia, permet à partir du nombre de threads par bloc, du nombres de registres par thread et de la quantité de mémoire partagée allouée par bloc de déterminer l'*occupancy*.

Ainsi, il est nécessaire de choisir la bonne combinaison de threads et de ressources allouées par bloc. Cependant, contrairement à ce qu'on pourrait croire, une occupancy maximale n'implique pas forcément des performances maximales. Volkov [124] démontre qu'on peut obtenir de meilleures performances avec un taux d'occupation des multiprocesseurs bas. C'est un point à prendre en considération quand les résultats paraissent anormalement faibles. Néanmoins, il est conseillé d'avoir au moins deux blocs à traiter par multiprocesseur.

3.6 Les "paralléliseurs" pour GPU

Bien que l'usage des GPU pour le calcul scientifique soit récent, plusieurs études ont déjà été conduites afin de simplifier la génération de code sur accélérateurs aux utilisateurs non expérimentés. Ces approches proposent une extension du langage C ou Fortran pour un portage en douceur sur les systèmes hétérogènes composés d'un GPP et d'un accélérateur GPU. Elles sont basées sur l'utilisation de directives spécifiant au compilateur les régions du code à décharger sur le GPU. On peut lister parmi ceux-là : HMPP [32], PGI Accelerator [129], Pathscale ENZO, HiCUDA [49], OmpSs [38], OpenACC...

Membarth et al. [76] évaluent cinq de ces outils sur une application d'imagerie médicale.

10. http ://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

L'environnement de développement Hybrid Multi-core Parallel Programming (HMPP) [32], proposé par la compagnie française CAPS, accompagne le programmeur dans la parallélisation de son code et son portage sur GPU ou FPGA. Leur approche est d'annoté les sources, codés en C ou en Fortran, avec des directives #pragma de façon similaire à OpenMP. Les déplacements de données entre GPP et GPU sont aussi exprimés à travers ces directives. HMPP est constitué d'un meta-compilateur apportant des indices d'optimisations à l'utilisateur et d'une bibliothèque d'exécution supportant la programmation en CUDA (NVidia) et CAL (AMD).

PGI Accelerator [129], une approche similaire de Portland Group (filiale à 100% de STMicroelectronics), est un compilateur pour langages C et Fortran employant CUDA comme langage de programmation GPGPU. Ainsi, seules les cartes compatibles CUDA sont supportées. Aussi, il n'est pas possible de fournir du code GPGPU codé à la main au compilateur car cela interférerait avec la parallélisation automatique des boucles dans les kernels.

La compagnie de compilateurs PathScale a récemment lancé un firmware pour GPGPU appelé ENZO. Bien que le modèle de programmation soit adapté à toutes architectures, il cible prioritairement les GPU Nvidia et/ou les GPP multi-cœurs. ENZO présente ses propres pilotes GPU permettant du calcul uniquement scientifique (pas d'usage graphique). Pathscale espère ainsi obtenir de meilleures performances que les pilotes Nvidia. Pour compenser le manque de permissivité d'optimisations bas-niveau à l'utilisateur, le modèle utilise des librairies optimisées et des mécanismes de raffinage automatique de performances.

HiCUDA [49], développé à l'Université de Toronto, est également un langage de programmation haut-niveau à base de directives pour programmer en CUDA. Un compilateur source-à-source traduit le programme C séquentiel, annoté de pragmas HiCUDA, en codes CUDA et GPP. Les directives proposées permettent au programmeur de spécifier les régions de code exécutées sur GPP ou GPU, la localisation des données (host, mémoire globale du GPU, mémoire partagée du GPU, mémoire constante du GPU, etc)...

StarSs [93] introduit des annotations de langages par des directives #pragma. Les annotations encapsulent les calculs dans des tâches et spécifient la direction de ses paramètres (input/output/inout) de telle sorte que les dépendances entre les tâches puissent être déterminées à l'exécution et l'algorithme exécuté dans un formalisme de flot de données. CellSs [14] était le premier support exécutif de StarSs disponible. Il a été implémenté pour les processeurs Cell. Ensuite a suivi SMPSs [94, 13] pour les architectures homogènes GPP multi-cœurs. Et finalement, GPUSs [11] pour les GPU Nvidia (les accélérateurs basés sur OpenCL n'étaient pas encore supportés). L'interaction de MPI avec SMPSs dans un modèle de programmation hybride [72] a ensuite été retenue pour composer OmpSs [10, 12, 38]. OmpSs adresse le problème de la programmation sur cluster hybride et accepte désormais les accélérateurs compatibles OpenCL. L'allocation mémoire et les transferts de données sur multi-GPU sont gérés de façon automatique. La parallélisation est incrémentale permettant une optimisation étape par étape. Le niveau d'optimisation choisi et exprimé à travers la multitude d'annotations apportées au code dépendra du niveau de portabilité attendu (au prix d'une perte de performance).

L'environnement accepte une multitude de kernels et permet à l'utilisateur de fournir ses propres kernels optimisés.

Un standard, basé sur le même principe de directives, est depuis quelques temps en développement. OpenACC, initialement développé par PGI, Cray et Nvidia avec l'assistance de CAPS, est basé sur le modèle de programmation de PGI Accelerator. Les quatre compagnies ont l'intention de collaborer avec le comité de développement du langage OpenMP afin d'unir les spécifications d'OpenACC avec une version étendue d'OpenMP permettant l'exploitation d'accélérateurs. L'API, décrivant une collection de directives de compilateur, permet de spécifier les boucles et les portions de code C, C++ ou FORTRAN à décharger d'un GPP vers un accélérateur en conservant une certaine portabilité à travers les systèmes d'exploitations, les GPP et les accélérateurs. Toutes les allocations de données, leurs transferts, l'initialisation des accélérateurs, leur activation et leur extinction sont implicitement gérées par l'API. Le modèle de programmation permet également à l'utilisateur de communiquer avec le compilateur à différents niveaux d'optimisation, de la localité des données au sein de la hiérarchie mémoire de l'accélérateur jusqu'à l'assistance au mapping de boucles ou de bouts de code spécifiques sur l'accélérateur.

Ce type d'approche a l'avantage de faciliter la transformation de codes séquentiels en codes parallèles. Cependant, deux arguments peuvent être retenus contre elle. Premièrement, pour espérer les meilleurs performances, l'algorithme de l'application implémentée sur une architecture parallèle doit être pensé parallèle dès sa conception. Deuxièmement, le compilateur n'est pas suffisamment intelligent à l'heure actuelle pour fournir une solution d'adéquation algorithme architecture optimale. En effet, Hernandez et al. [54] rapportent un facteur deux entre CUDA et HMPP / PGI pour l'implémentation de l'application High-Order Multi-scale Modeling Environment (HOMME). Similairement, pour trois différentes tailles de volumes, Membarth et al. [76] démontrent que l'implémentation d'un algorithme de recalage d'images 2D/3D avec CUDA présente de meilleures performances qu'avec HMPP, PGI et RapidMind. Cependant, l'écart se rétrécie entre CUDA et HMPP pour les gros volumes 512 x 512 x 378 et HMPP produit même du code plus performant qu'un code OpenCL optimisé à la main.

Toute la question est de savoir si nous recherchons à :
- tirer les meilleures performances du matériel exploité : une démarche d'adéquation algorithme architecture (AAA) est alors nécessaire et l'intervention d'un expert s'avère être le meilleur choix.
- accélérer une application en portant son implémentation séquentielle sur un accélérateur : dans ce cas, la facilitée de programmation apportée par ces outils impacte avantageusement le temps de développement.

Nous nous retrouvons dans la même situation qu'à l'époque de l'avènement des premiers compilateurs. Coder en assembleur apportait de meilleures performances mais coder dans un langage compilé permet un développement plus rapide avec des performances raisonnables.

Chapitre 4

Accélération de l'application de granulométrie

Sommaire

4.1 La granulométrie 42
4.2 Travaux existants 44
4.3 Optimisations de l'algorithme de granulométrie 45
4.3.1 La forme de l'élément structurant 45
4.3.2 La taille de l'élément structurant 46
4.3.3 Les érosions redondantes 47
4.3.4 Les allocations mémoires 47
4.3.5 Le codage de l'information 50
4.4 Performances crêtes et facteur limitant sur GPU 52
4.5 Implémentation de l'algorithme optimisé sur GPU 54
4.5.1 La quantité de voxels traités par thread GPU 55
4.5.2 Les stratégies d'implémentation mémoire 58
4.5.3 Le cas d'une implémentation en mémoire partagée 68
4.5.4 Sommation des voxels d'intérêt sur tout le volume 69
4.6 Résultats expérimentaux 72
4.6.1 Station de travail 72
4.6.2 Accélération CPU 72
4.6.3 Accélération GPU 75
4.6.4 Investigation des variations de performances sur GPU 78
4.6.5 Gain du GPU sur le CPU 86
4.7 Conclusion 88
4.7.1 Gains en performances 89

4.7.2 Retour d'expérience sur le portage d'algorithmes sur GPU . . 89
4.7.3 Degré d'optimisation de l'algorithme CPU 89
4.7.4 Vers des gains plus audacieux ? 90

Dans ce chapitre, nous étudierons le portage d'un algorithme de traitement d'image couramment utilisé dans le domaine de la science des matériaux : la granulométrie. Nous commencerons par décrire l'algorithme de granulométrie (4.1) puis étudier la liste des travaux existants dans ce domaine (4.2). Nous présenterons ensuite les optimisations de l'algorithme de granulométrie (4.3) puis son implémentation GPU (4.4). Finalement, nous définirons les facteurs responsables des variations de performance du GPU (4.5).

4.1 La granulométrie

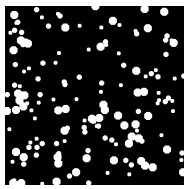
La granulométrie est l'étude de la distribution statistique de la taille des éléments finis d'une population. Autrement dit, son but est de déterminer la répartition des tailles d'objets dans une image [74, 26]. Le moyen de procéder consiste à comptabiliser les éléments en commençant par les plus petits et en terminant par les plus gros. Une analogie physique du processus serait un tamisage itératif du volume avec une taille de maille croissante. En pratique, des opérations de morphologie mathématiques [105] (Annexe A) sont employées pour effectuer ce tamisage. Plus précisément, ce sont des opérations d'ouverture qui sont utilisées. La taille de l'élément structurant détermine la taille du maillage du tamis. En effet, après une ouverture, les objets dont la taille est inférieure à l'élément structurant sont éliminés. Ainsi, l'algorithme de granulométrie effectue des ouvertures, de façon itérative, avec une taille d'élément structurant croissante. Après chaque ouverture, on comptabilise le nombre de pixels d'intérêt (dans notre cas, les pixels blancs) restants dans l'image. S'il en reste, on réitère en incrémentant la taille de l'élément structurant. Sinon, l'algorithme a terminé et nous disposons d'un tableau contenant pour chaque ouverture le nombre de pixels dans l'image (Algorithme 1).

A partir de ces données, une courbe présentant le nombre de pixels d'intérêt (dans notre cas, les pixels blancs) restant en fonction de la taille de l'élément structurant est tracée. Cette courbe est appelée courbe granulométrique (Figure 4.1). Sa dérivée représente la différence du nombre de pixels entre chaque itération. A partir du tracé de la dérivée, on déduit facilement la taille des objets prédominants dans l'image : l'abscisse correspondant au maximum global de la courbe représente la taille de l'élément structurant qui a supprimé le plus grand nombre de pixels dans l'image après ouverture. La taille d'objet qui occupe la majeure partie de l'image est donc celle de la taille de l'élément structurant susmentionné.

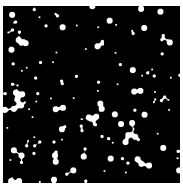
Algorithme 1 Pseudo-code de l'algorithme de granulométrie

Entrées: Un volume binaire $VolBin$ à traiter
Sorties: Le tableau de données $TabVoxelCounter$ pour la courbe granulométrique

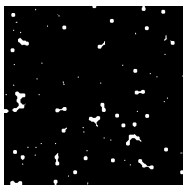
1: $NbIteration \leftarrow 0$
2: **tantque** $VoxelCounter \neq 0$ **faire**
3: $\quad VoxelCounter \leftarrow 0$
4: $\quad NbIteration \leftarrow NbIteration + 1$
5: $\quad ElemStruct \leftarrow GenereElemStruct(NbIteration)$
6: $\quad VoxelCounter \leftarrow Ouverture(VolBin, ElemStruct)$
7: $\quad TabVoxelCounter[NbIteration] \leftarrow VoxelCounter$
8: **fin tantque**
9: $AfficherGraphique(TabVoxelCounter)$



(a) taille 4

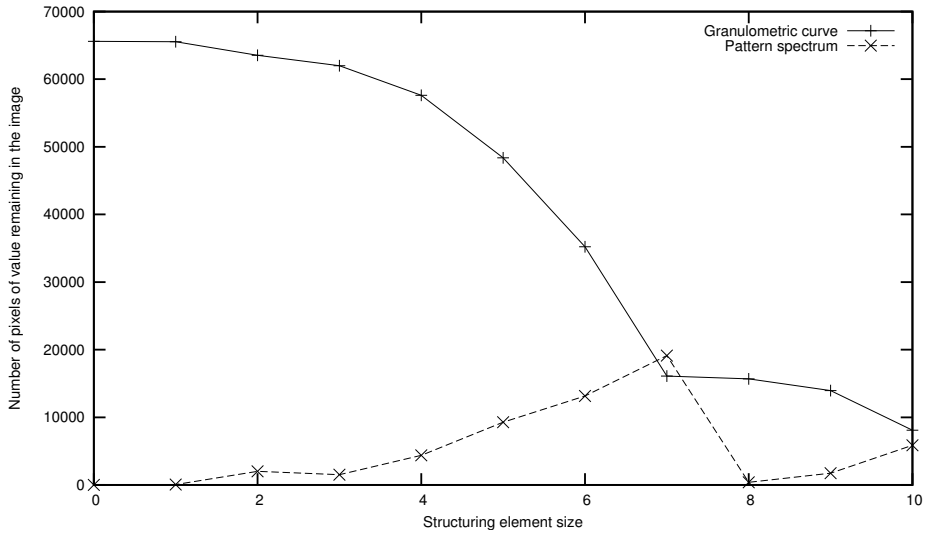


(b) taille 6



(c) taille 8

(d) taille 10



(e) La courbe granulométrique associée aux images précédentes. Le point culminant de sa dérivée indique que les objets de taille 7 sont prédominants dans l'image.

FIGURE 4.1 – Un échantillon (a), (b), (c), (d) d'images sortantes après une ouverture pour différentes tailles de l'élément structurant ; les pixels blancs sont les pixels d'intérêt. Quand l'image sortante ne contient plus de pixels blancs, l'algorithme a terminé. La Figure (e) représente l'allure de la courbe granulométrique.

4.2 Travaux existants

Le concept de la granulométrie a été introduit par G. Matheron en 1967 [75] comme un nouvel outil pour étudier les milieux poreux. La granulométrie constitue un des outils les plus intéressants et les plus versatiles pour l'analyse morphologique des images. Elle peut être appliquée dans un large éventail de traitements : de l'extraction de points particuliers, en passant par la caractérisation de la texture, l'estimation des tailles d'objets à la segmentation d'images. Cependant, à cause du temps d'exécution élevé de ses longues séquences d'ouvertures par des éléments structurant de taille grandissante, sa popularité est faiblement établie dans la communauté du traitement de l'image.

L'érosion et la dilatation étant les opérations morphologiques de base [105, 51], une quantité considérable de littérature a été produite, autour du sujet, rapportant des techniques améliorant la vitesse d'exécution de ces opérateurs fondamentaux [90, 24, 27, 80].

Pour des images en niveau de gris, Van Herk a montré comment effectuer des érosions et des dilatations avec un élément structurant linéaire (horizontal, vertical ou diagonale) de longueur arbitraire en utilisant seulement 3 opérations min/max par pixel [120]. Soille et al. [110] étendent ses travaux en adaptant l'utilisation d'éléments structurant linéaires quel que soit leur orientation. Comptons également parmi les travaux sur image en niveaux de gris, les travaux de Vincent [123, 122] et Van Droogenbroeck [119]. Haralick [50] pour sa part constitue une référence concernant les opérations d'ouverture récursives.

Toutefois, nos travaux se focalisent sur le traitement d'images binaires noir/blanc. Pour cela, ce sont d'autres travaux qui ont attiré notre attention. Vincent [121] présente l'utilisation d'éléments structurant de forme arbitraire pour le traitement optimisé d'opérations de morphologie mathématiques sur des images binaires 2D. Nikopoulos et al. [81] l'adapte pour des images binaires 3D. Ces techniques font cependant appel à des traitements à passes multiples. Trois passes sont nécessaires pour effectuer une opération d'érosion ou de dilatation selon la méthode proposée par Nikopoulos et al. [81].

Pour résumer, la majorité des optimisations de l'application de granulométrie sont étudiées pour des images en niveaux de gris et/ou pour des éléments structurants dont la forme est linéaire (et parfois rectangulaire) sur des architectures séquentielles. Certains travaux ont portés l'application de granulométrie sur GPU [70, 59].

Cependant, dans notre cas précis, une image binaire 3D traitée par un octaèdre comme élément structurant (forme la plus proche des éléments à détecter) aucune étude n'a été effectuée à notre connaissance. Pour cela, les choix d'implémentation de l'algorithme de granulométrie sur GPU que nous avons faits sont simples. Toutefois, notre implémentation présente une innovation qui est le codage de l'information (les voxels) sur un 1 bit. Cette stratégie augmente le débit et réduit la taille des espaces mémoires alloués (important alors que les GPU utilisés contenaient entre 256 et 512 Mo de mémoire embarquées au début de cette thèse).

4.3 Optimisations de l'algorithme de granulométrie

Suite à la description de l'algorithme, nous allons nous pencher sur son implémentation et les choix d'optimisation. Ces derniers sont répartis en deux volets : algorithmique (ci-présent) et architectural (dans la section 4.2 suivante). Dans un premier temps, nous présenterons donc les optimisations algorithmiques puis, dans un deuxième temps, les optimisations liées à l'architecture d'implémentation.

A noter que l'algorithme présenté est décrit en 2D afin de simplifier l'explication. Il sera par la suite adapté pour être appliqué sur des volumes 3D. Le principe reste le même qu'en 2D, seulement l'élément structurant prend une dimension supplémentaire (dimensions x,y et z).

4.3.1 La forme de l'élément structurant

En filtrage, pour certaines formes de masque, il est possible de minimiser la redondance des calculs. Si le masque a une forme rectangulaire (2D), il est possible de le séparer en deux filtres 1D. Si le masque a une forme parallélépipédique (3D), il est possible de le séparer en trois filtres 1D. Aussi, en 3D il peut avoir une forme de masque non séparable suivant un plan 2D mais cette forme se réplique suivant la dernière dimension, il est donc séparable en deux filtres : un filtre 2D inséparable et un filtre 1D.

Démonstration mathématique pour un masque 2D décomposable :

$$Masque = \begin{bmatrix} A \cdot a & A \cdot b & A \cdot c \\ B \cdot a & B \cdot b & B \cdot c \\ C \cdot a & C \cdot b & C \cdot c \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot \begin{bmatrix} a & b & c \end{bmatrix} \tag{4.1}$$

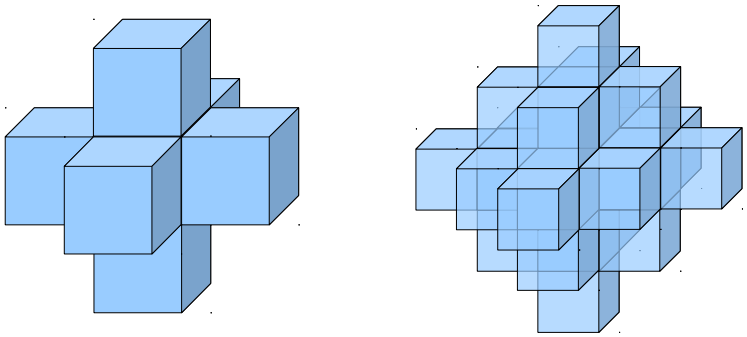
L'équation de convolution de l'Image par le Masque est alors décomposée en deux étapes :

$$\begin{aligned} Image * \begin{bmatrix} A \cdot a & A \cdot b & A \cdot c \\ B \cdot a & B \cdot b & B \cdot c \\ C \cdot a & C \cdot b & C \cdot c \end{bmatrix} &= Image * \left(\begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot \begin{bmatrix} a & b & c \end{bmatrix} \right) \\ &= \left(Image * \begin{bmatrix} A \\ B \\ C \end{bmatrix} \right) * \begin{bmatrix} a & b & c \end{bmatrix} \end{aligned} \tag{4.2}$$

Cette convolution en deux étapes est illustrée à la Figure 4.2.

Dans notre cas, l'élément structurant est un octaèdre. Ce choix est justifié par la forme sphérique des objets que le SIMaP souhaite détecter dans les images. L'élément structurant

FIGURE 4.2 – Séparation de masque pour une convolution d'image.



(a) taille 1 (b) taille 2

FIGURE 4.3 – Différentes tailles de l'élément structurant en forme de croix 3D qui est un filtre inséparable.

de taille 1 est représenté par une croix en 3 dimensions de connectivité 6 (Figure 4.3a). Ce n'est donc pas un filtre séparable et cette optimisation ne peut avoir lieu.

4.3.2 La taille de l'élément structurant

Une érosion/dilatation par un élément structurant de taille $m = n \times x$ fourni le même résultat qu'une succession de n érosions/dilatations par un élément structurant de taille x. A noter qu'un élément structurant de taille x est la succession de x dilatations d'un voxel par l'élément structurant de taille 1 (Figure 4.3). 1 érosion par un élément structurant de taille m correspond donc à m érosions par un élément structurant de taille 1.

Un élément structurant de taille 1 présente 6 voxels voisins (les voisins directs du voxel concerné) (cf. Figure 4.3a). Tandis qu'un élément structurant de taille 2 présente 24 voisins (cf. Figure 4.3b), deux érosions/dilatations successives par un élément structurant de taille 1 implique $2 \times 6 = 12$ opérations logiques. Une érosion/dilatation par un élément structurant de taille 2 nécessite alors 2x plus d'opérations ET/OU logiques qu'une succession de deux érosions/dilatations par un élément structurant de taille 1. Le nombre d'accès mémoire est également bien supérieur dans le premier cas (cf. Tableau 4.1).

	n : taille de l'élément structurant	# accès mémoire lecture	# accès mémoire écriture	# opérations logiques
n × ElemStructTaille(1)	1	7	1	6
	2	14	2	12
	3	21	3	18
1 × ElemStructTaille(n)	1	7	1	6
	2	25	1	24
	3	63	1	62

TABLE 4.1 – Quantité d'accès mémoire et d'opérations logiques en fonction de la stratégie choisie pour les trois premières taille de l'élément structurant.

4.3.3 Les érosions redondantes

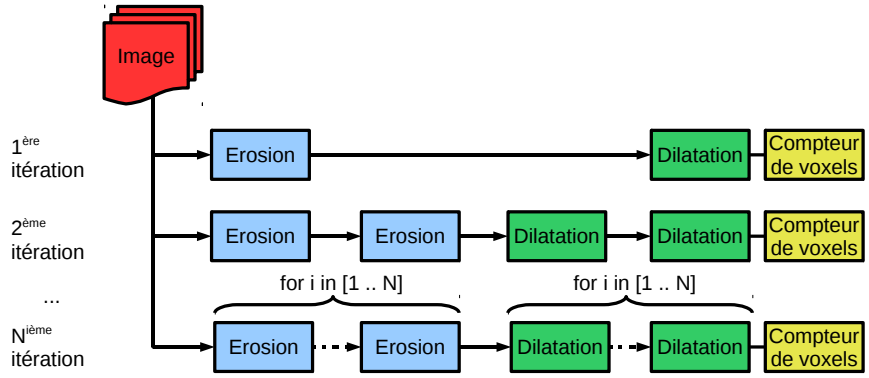
Une ouverture par un élément structurant de taille n est la succession d'une érosion et d'une dilatation par l'élément structurant de taille n. Une alternative est la succession de n érosions puis de n dilatations par l'élément structurant de taille 1. Ainsi, d'une ouverture à la suivante, certains traitements sont redondants (Figure 4.4a). En effet, les $(n-1)$ érosions calculées à l'itération $(n-1)$ sont recalculées à la $n$ième itération. Afin d'éviter cette redondance de calcul, le résultat de la $n$ième érosion doit être sauvegardé dans un espace mémoire tampon afin d'être réutilisé pour le calcul de la $(n+1)$ième érosion à l'itération suivante (Figure 4.4b).

$$Gain \approx \frac{n\ erosions + n\ dilatations}{1\ erosion + n\ dilatations} \approx \frac{2n}{n+1} \underset{n\ grand}{\rightarrow} 2$$

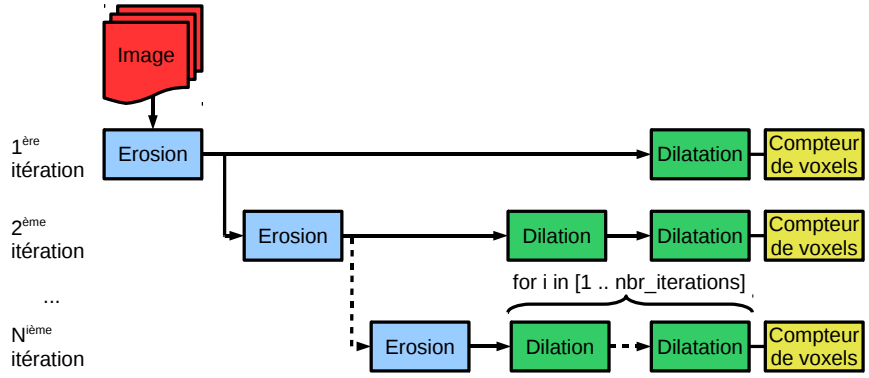
4.3.4 Les allocations mémoires

Dans la suite, nous désignerons un "espace mémoire" par un "buffer". Une implémentation naïve allouerait un buffer pour chaque tâche lancée. Considérant l'algorithme de granulométrie non-optimisé (Figure 4.4a), $2n+1$ buffers seraient alors alloués : l'image de départ, les n érosions et les n dilatations (Figure 4.5a). En supprimant le calcul des érosions redondantes (Figure 4.4b), à la $n$ième itération, seul une érosion sur les n initiales sont exécutées poursuivies par n dilatations. Soient, $n+1$ tâches lancées séquentiellement (Figure 4.5b).

Cependant, les dilatations s'enchaînent séquentiellement. Comme l'image fournie en entrée de la fonction de dilatation n'est plus réutilisée par la suite, l'usage de "double buffers" alternant le buffer source et le buffer résultat semble approprié. Ainsi, seuls deux buffers sont alloués. Or, le résultat de l'érosion doit être sauvegardé pour le calcul de l'érosion à la prochaine itération. Pour cela, un buffer supplémentaire est nécessaire. Nous sommes donc partis sur



(a) non-optimisé : redondance des calculs.



(b) optimisé : sauvegarde de la dernière érosion dans un espace mémoire tampon pour la prochaine itération.

FIGURE 4.4 – Optimisation de la séquence d'opérations d'érosion/dilatation de l'algorithme de granulométrie. Chaque tâche comporte une couleur : rouge pour la lecture du volume, bleue pour l'érosion, verte pour la dilatation et jaune pour le comptage de pixels d'intérêts

une solution à 3 buffers.

Listing 4.1 présente le pseudo-code de la solution accompagnée d'une démonstration par l'exemple sur les 5 premières itérations du processus (Figure 4.5c). Une gestion efficace des pointeurs vers les 3 espaces mémoires suffit à éviter des copies mémoires superflues.

L'usage de mini-kernels (érosion et dilatation) prononce le découpage de l'algorithme en phases de calcul distinctes. A la fin de l'exécution d'un kernel, avant de rendre la main au CPU, la fin des calculs et des accès mémoires pour tous les SM est assurée. En d'autres termes, la terminaison d'un kernel induit une synchronisation implicite des données entre les SM. On est donc sûr que toutes les données résultat du kernel $n-1$ ont été sauvegardées dans l'espace mémoire cible avant de lancer le prochaine kernel n.

Listing 4.1 – Pseudo-code de l'algorithme de granulométrie. Les transferts mémoires CPU $\leftrightarrow$ GPU ne sont pas marqués pour simplifier le code. Les mini-kernels (érosion et dilatation) permettent une synchronisation implicite des données.

```
/* initialize */
opening_counter = 1;
do{
    /* erosion */
    erosion3D<<<grids, blocks>>>(gmem2, gmem1);
    temp_erode_gpu = gmem2;

    /* dilatations */
    if ( opening_counter == 1 )
    {
        dilation3D<<<grids, blocks>>>(gmem3, gmem2);
        gmem2 = gmem1;
    }else{
        // 1st iteration
        dilation3D<<<grids, blocks>>>(gmem3, gmem2);
        temp_dilate_gpu = gmem3;
        gmem3 = gmem1;
        gmem2 = temp_dilate_gpu;
        // n-1 iterations
        for(unsigned int i=1; i<opening_counter; i++)
        {
            dilatation3D<<<grids, blocks>>>(gmem3, gmem2);
            temp_dilate_gpu = gmem3;
            gmem3 = gmem2;
            gmem2 = temp_dilate_gpu;
        }
    }
    nb_pixels = get_nb_pixels_left();
    gmem1 = temp_erode_gpu;
```

```
    opening_counter++;
}while(nb_pixels = 0);
```

$$\boxed{Gain\ memoire \approx \frac{1\ volume\ initial + 1\ erosion + n\ dilatations}{3} \approx \frac{n+2}{3}}$$

4.3.5 Le codage de l'information

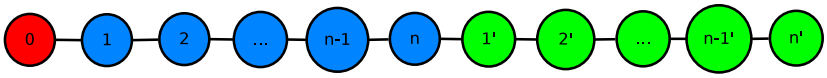
Bien qu'une image binaire présente moins d'informations qu'une image en nvg, ces deux images sont généralement codées sur la même unité d'allocation mémoire : l'octet. Il existe pourtant un format de fichier pour sauvegarder des images noir et blanc en binaire : PBM. Seulement, la majorité des logiciels codent l'information sur l'unité d'allocation mémoire la plus petite : l'octet. Ainsi, la plus petite unité de codage de l'information pour le logiciel de traitement d'images ImageJ [1] est sur 8-bits : 0 pour noir et 255 pour blanc. La rareté des logiciels interprétants les images codées sur 1 bit peut être justifiée par la quantité suffisante d'espace mémoire dont dispose les ordinateurs aujourd'hui.

Cependant, la première carte GPU sur laquelle nous avons débuté le portage de l'algorithme, la GeForce 9500 GT, ne disposait que de 256 Mo de mémoire globale. Sachant que nous travaillions sur des volumes d'au moins 512^3 voxels, c'est-à-dire 134 Mo environ (codage 8-bits), il suffit d'allouer un espace pour l'image d'entrée et un autre pour l'image résultat pour déborder des ressources mémoire disponibles sur la carte GPU. Pour cela, nous avons choisi de binariser l'image en noire et blanc (passage du codage d'un voxel sur 1 octet à 1 bit). Cela a permis de réduire par 8 la quantité de mémoire allouée. Ceci n'est pas négligeable quand nous savons que la taille des volumes de données à traiter ne cesse d'augmenter : la taille des données actuellement recueillie en tomographie avoisine facilement le Go.

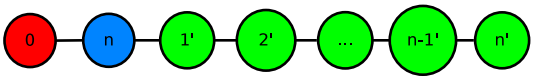
Ce choix de stockage de l'information a également des répercutions positives sur le débit sortant des voxels résultats. Néanmoins, une manipulation des voxels à l'échelle du bit est plus complexe. D'ailleurs, il est bon d'être familiarisé avec certaines notions comme l'endianness. La bonne marche à suivre est décrite dans la partie architecturale dédiée à l'implémentation GPU bien qu'elle soit également adaptée au CPU (4.5.1).

$$\boxed{Gain \approx \frac{8\ voxels\ par\ octet}{1\ voxel\ par\ octet} \approx 8}$$

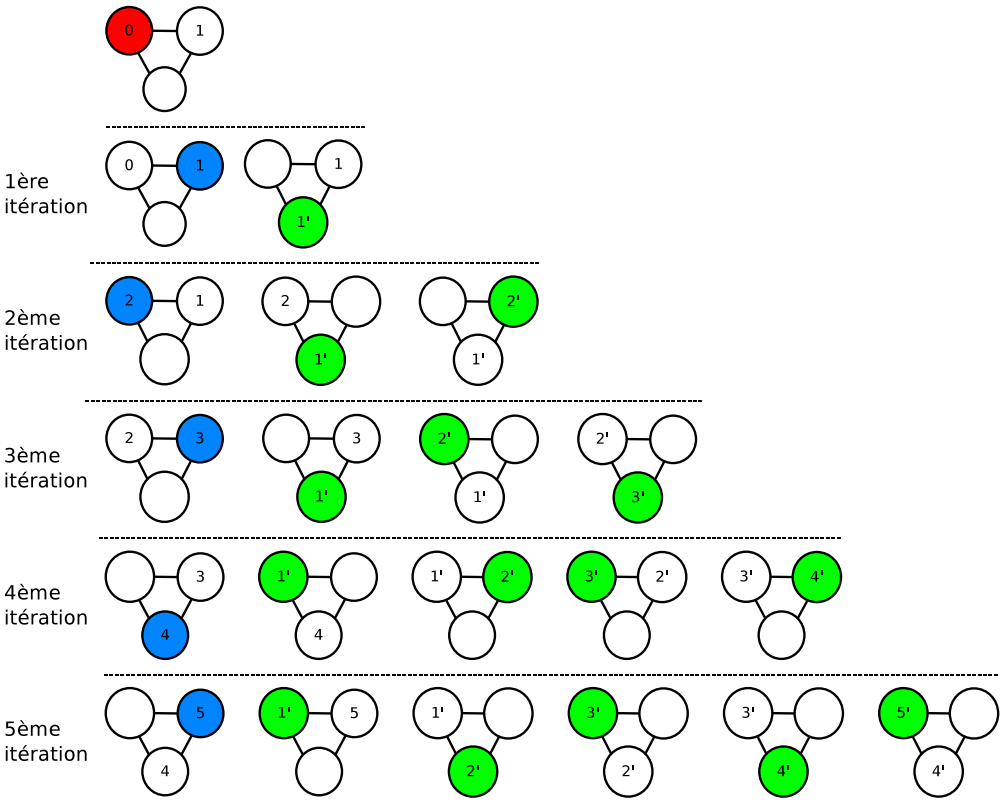
1. http ://rsb.info.nih.gov/ij/



(a) Allocation naïve de buffers pour l'algorithme non-optimisé (Figure 4.4a) : l'image d'origine, les n érosions suivies des n dilatations.



(b) Allocation naïve de buffers pour l'algorithme optimisé (Figure 4.4b) : l'image d'origine, la nième érosion suivie des n dilatations.



(c) Illustration de la gestion des 3 buffers pour les 5 premières itérations de l'algorithme. A chaque itération, le résultat de l'érosion est sauvegardé dans un buffer (en bleu) puis les 2 autres buffers sont utilisés comme un double-buffer entrée/sortie des n dilatations suivantes (en vert). A chaque étape, deux buffers sont utilisés : un (non colorié) avec les données d'entrée et l'autre (colorié) avec les données de sortie. Les numéros simples tels que (1), (2), (3), ... correspondent à des buffers contenants les résultats des érosion tandis que les buffers de numérotation (1'), (2'), (3'), ... contiennent les résultats des dilatations.

FIGURE 4.5 – Etapes successives d'optimisation du nombre de buffers alloués pour l'algorithme de granulométrie

4.4 Performances crêtes et facteur limitant sur GPU

Selon Mickevicius [88], il existe trois facteurs de performance limitants à une implémentation GPU :

- la bande passante mémoire
- la latence des accès mémoire
- le débit en instructions

Et il existe trois façons de définir le(s) facteur(s) limitant(s). Ces trois méthodes, présentées dans un ordre croissant de sophisticité, apportent des résultats de fiabilité croissante :

une étude algorithmique , à partir des besoins mémoires et arithmétiques, sauf qu'elle sous-estime le nombre d'instructions et les accès mémoires.

le profiler , à partir de compteurs d'accès mémoires et d'exécution d'instructions, est plus précis que le point précédent, mais moins efficace que le dernier point.

la modification du code source en supprimant la partie complémentaire (mémoire seule ou arithmétique seule) afin de mesurer un facteur à la fois. Cela représente le meilleur des cas, mais ce n'est pas possible pour tous les codes (surtout quand le flot de contrôle dépend des résultats de données).

Notre code étant relativement basique, la dernière approche, accompagnée du profiler, nous apportera les meilleurs résultats.

Pour qu'un facteur soit limitant, c'est que les performances attendues ne sont pas au rendez-vous. Pour cela, avant de déceler le(s) facteur(s) limitant(s), il faut d'abord avoir une estimation des performances théoriques atteignables. A partir du Tableau 4.2 et des données fournies en annexe B.1, nous pouvons calculer certaines valeurs théoriques de référence : le débit d'instructions maximal $Throughput\ of\ Native\ Arithmetic\ Instructions \times (GPU\ clock\ speed \times 2) \times \#SM$ et la bande passante mémoire maximale $(Memory\ Bus\ Width/8) \times (Memory\ Clock\ rate/2)$.

Un algorithme dont le ratio $\frac{debit}{bandepassante}$ est proche de celui offert par les GPUs a de bonnes chances d'obtenir des performances intéressantes. D'ailleurs, les ratios calculés dans le Tableau 4.3 montrent que les GPUs sont plus adaptés aux algorithmes arithmétiquement gourmands.

L'algorithme de granulométrie étant essentiellement composé d'opérations de calcul (peu de contrôle), une première estimation serait qu'il y a autant d'opérations logiques que d'accès mémoire : 7 accès mémoire et 6 opérations logiques (Equation 4.6) plus 2 accès mémoires, 2 opérations logiques et 4 décalages binaires (Equation 4.7 et 4.8), soit un total de 9 accès mémoire, 8 opérations logiques et 4 décalages binaires. Le ratio est donc proche de 1 et le facteur limitant serait logiquement la bande passante mémoire ou la latence. Bien entendu, les déductions de ce calcul restent hypothétique puisqu'elles ne prennent pas en considération les instructions de flot de contrôles, le calcul des adresses mémoires ou bien les types d'accès

	Compute capability				
	1.0 1.1 1.2	1.3	2.0	2.1	3.0
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192
64-bit floating-point add, multiply, multiply-add	1	1	16	4	8
32-bit integer shift	8	8	16	16	32
Logical operations	8	8	32	48	160

TABLE 4.2 – Débit de certaines instructions arithmétiques (opérations / cycle horloge / multiprocesseur). (source : NVidia)

	débit (Gflops)	bande passante mémoire globale (Go/s)	ratio
GeForce GTX 285	$8 \times (1.48 \times 2) \times 30$ **= 710.4**	$(512/8) \times (1242/2)$ **= 158**	**4,4**
GeForce GTX 480	$32 \times (1.40 \times 2) \times 15$ **= 1 344**	$(384/8) \times (1848/2)$ **= 177**	**7,5**
Quadro 4000	$32 \times (0.95 \times 2) \times 8$ **= 486.4**	$(256/8) \times (1404/2)$ **= 89.6**	**5,4**

TABLE 4.3 – Débit des processeurs GPU pour des instructions fp32 (similaire aux instructions d'opération logique), bande passante de la mémoire globale et ratio de ces deux caractéristiques pour différents GPUs.

mémoire : mémoire globale, mémoire partagée, registres, etc.

4.5 Implémentation de l'algorithme optimisé sur GPU

Avant de présenter notre implémentation GPU, présentons les arguments qui font de cet algorithme un choix adapté au GPGPU. Ci-dessous la liste des éléments permettant de positionner la granulométrie parmi les applications favorables au portage sur GPU :

- Dans le modèle d'exécution *Single Instruction Multiple Threads* (SIMT) de CUDA, chaque warp exécute une seule et unique instruction. Il faut donc éviter les divergences de chemin d'exécution entre les *threads* dans un *warp*. Or, que ce soit pour une érosion ou une dilatation, les opérations de morphologie mathématiques utilisent exclusivement les opérateurs logiques ET (érosion) et OU (dilatation). En d'autres termes, un seul type d'opérateur logique est utilisé sur tous les voxel *i.e.* une seule instruction est issue pour toutes les données et très peu d'instructions de contrôle sont présentes. C'est donc un algorithme avec peu d'instructions divergentes.
- Le rapport élevé $\frac{operations\ arithmetiques\ et\ logiques}{operations\ de\ controle}$ est adapté à l'architecture des GPU comparée à celle des CPU (Figure **??**).
- L'interdépendance des données est connu à priori : la forme de l'élément structurant détermine les dépendances entre les voxels. Il est donc aisé de gérer cet aspect primordial dans toute programmation parallèle.
- L'algorithme est composé d'une succession de tâches simples : érosions et dilatations. Chaque tâche constitue un kernel GPU. Après chaque tâche, suite à la terminaison du kernel, les SM sont synchronisés de façon implicite avant de rendre la main au CPU. Cette synchronisation matérielle permet d'assurer la cohésion des données résultats. Les données initialement réparties sur les multiprocesseurs par découpage en blocs sont ainsi re-synchronisées après traitement.
- Le traitement des données ne suit pas un ordre particulier *i.e.* l'accès aux données en mémoire peut être arbitraire. Il est donc possible de les traiter dans la configuration la plus avantageuse, c'est-à-dire séquentiellement, afin d'avoir des accès "coalesced" à la mémoire globale du GPU.

Malgré la scalabilité des données offerte par le modèle de programmation CUDA, le développement sur GPU est étroitement lié à l'architecture, nous sommes donc confrontés aux contraintes de l'*Adéquation Algorithme Architecture* (AAA). Afin qu'une augmentation du nombre de *CUDA Cores* assure un gain en performance linéaire, il faut pouvoir appuyer son implémentation sur des caractéristiques architecturales inter-générationnelles communes.

Sans *road map*, l'évolution de l'architecture des cartes graphiques, imprévisible pour le programmeur, rend l'implémentation d'une solution GPU scalable en performance difficile. Le programmeur peut tenter de conjoncturer les caractéristiques matérielles qui devraient rester figer. Cependant, à terme, si les prévisions du programmeur sont fausses, ses efforts de

portage s'avéreront inutiles pour les prochaines générations de cartes GPU. C'est d'ailleurs ce qui s'est passé pour plusieurs implémentations GPU lors du passage de la GT200 à la Fermi.

Dans ce volet, nous présenterons les choix d'optimisation liés aux caractéristiques matérielles connues. L'implémentation proposée tente de concilier au mieux les différentes architectures de GPU existantes. Cette approche théorique basée sur des connaissances approfondies en architecture des processeurs sera complétée dans la section 4.7 par une approche pratique *i.e.* des mesures de performances sur les configurations systèmes existantes au laboratoire.

4.5.1 La quantité de voxels traités par thread GPU

Chaque voxel étant codé sur 1 bit (4.3.5), tentons maintenant de déterminer le nombre de bits (et donc de voxels) que doit traiter chaque thread GPU. Les points à prendre en considération pour effectuer ce choix sont :

- la taille des opérandes : les Unités Arithmétiques et Logiques des cartes NVidia peuvent effectuer des opérations sur 32-bits ou 64-bits. Par contre, les opérateurs logiques ET/OU ne prennent que des opérandes entiers 32-bits [86]. Choisir une taille d'opérande inférieur à 32-bits est donc sous-optimal : cela nécessiterait une conversion implicite des données (instructions supplémentaires).
- les tailles de banc de la mémoire partagée : Un accès dans un banc mémoire s'effectue sur un mot de 32-bits. Pour un groupe de threads (demi-warp ou warp), un accès à la mémoire partagée est optimal quand chaque thread accède à un banc distinct. Si deux threads accèdent au même banc mémoire, les deux accès sont sérialisés. Pour des accès optimisés, il faut donc éviter que les threads d'un même warp accèdent simultanément (en lecture ou écriture) à des voxels disposés dans le même banc mémoire.
- les accès "coalesced" à la mémoire globale (4.5.2) : une condition nécessaire (quelque soit la génération de la carte graphique) est que chaque thread accède à 4, 8 ou 16 octets. A partir des générations de carte Fermi [82], le nombre de transactions mémoire augmente au delà de 4 octets par thread. Il n'y a donc aucun bénéfice à augmenter le nombre d'octets accédés par thread. En fait, cela pourrait même desservir les performances puisque chaque transaction supplémentaire provoquera un délai d'attente à l'exécution d'un demi-warp : il faut détenir les données avant de pouvoir les traiter.

Le dénominateur commun à tous ces points est la valeur de 32-bits (4 octets). Traiter 32-bits par threads est donc la meilleure solution, fournissant un débit de 32 voxels / thread. Cela implique cependant une contrainte : l'image doit avoir une dimension x multiple de 32 voxels.

Dans les deux paragraphes suivants nous allons traiter deux aspects de l'implémentation des opérations d'érosion et dilatation avec un codage de l'information binaire sur 1-bit :

- L'*endianness i.e.* l'organisation des octets en mémoire. Nous verrons que certaines organisations peuvent inverser les octets et provoquer un désordre dans la séquence des

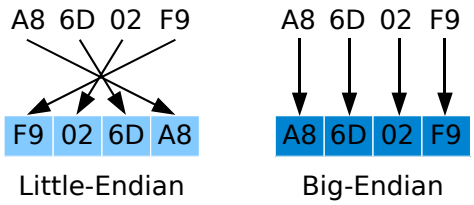


FIGURE 4.6 – Organisation dans la mémoire d'un mot de 4 octets selon l'endianness.

voxels lus.
- Comment effectuer une érosion/dilatation sur 32 voxels simultanément ? Nous verrons les manipulations de bits nécessaires à cette opération.

L'*endianness* : l'ordonnancement des octets en mémoire

A noter que l'endianness (anglicisme traduit par "boutisme") des architectures CPU/GPU doit être prise en compte pour ne pas confondre l'emplacement des voxels dans la mémoire. L'orientation little-endian (ou petit-boutiste) inverse la position des octets en mémoire (Figure 4.6).

Listing 4.2 indique comment bien entreprendre la binérisation de l'image d'origine dans ces conditions. Ainsi, les voxels du mot lu sont organisés séquentiellement dans le bon ordre et le résultat des décalages binaires est juste.

Listing 4.2 – Fonction de binarisation de l'image RAW en entrée

```
void uchar_to_binary( unsigned char *binary_out,
                      unsigned char *uchar_in,
                      int mem_size)
{
        int i=0, j=0;
        while( i < mem_size)
        {
                binary_out[j]=0;
                for (int k = 0; k < 8; i++,k++)
                        binary_out[j] |= uchar_in[i] & (1<<k);
                j++;
        }
}
```

Par exemple, si la séquence suivante devait être traitée :

$$00111101 \quad 11110001 \quad 10011010 \quad 01011000 \tag{4.3}$$

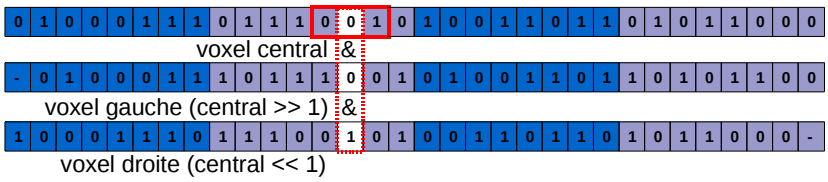


FIGURE 4.7 – Pour effectuer les opérations ET/OU logiques entre le voxel central et les voxels latéraux, on utilise un décalage binaire. L'illustration montre l'exemple d'un voxel (codé sur un bit) cependant les 31 autres voxels contenus dans le mots de 4 octets sont traités simultanément de la même façon.

Sa lecture en petit-boutiste donnerait :

$$01011000 \quad 10011010 \quad 11110001 \quad 00111101 \tag{4.4}$$

Du coup, en binarisant comme indiqué dans Listing 4.2, sa lecture donnera plutôt le rendu final suivant :

$$00011010 \quad 01011001 \quad 10001111 \quad 10111100 \tag{4.5}$$

Si vous regardez bien, vous vous apercevrez que la ligne 4.3 n'est en fait que le reflet miroir de la ligne 4.5.

Effectuer une opération d'érosion/dilatation sur un vecteur

Voyons maintenant comment manipuler des mots de 32-bits pour effectuer les opérations de ET et OU logique sur un voisinage de voxels. Considérons l'équation d'érosion par l'élément structurant de la Figure 4.3a :

$$resultat \;=\; haut \;\&\; bas \;\&\; gauche \;\&\; central \;\&\; droite \;\&\; entrant \;\&\; sortant \tag{4.6}$$

Lorsqu'un voxel est codé sur un octet (unsigned char), l'érosion est facilement implémentée. Le langage de programmation CUDA (extension du langage C) dispose de variables de type "unsigned char". La manipulation des voxels est donc facilitée pour le programmeur.

Cependant, lorsqu'un voxel est codé sur un bit, l'implémentation devient un peu plus compliquée. L'accès aux voxels voisins latéraux, droite et gauche, nécessite l'usage un décalage binaire, gauche et droite respectivement (Figure 4.7).

L'accès aux voxels latéraux de l'équation 4.6 est donc modifiée de sorte à être appliquée à un codage binaire avec 32-bits traités simultanément par un thread :

$$gauche \;=\; ((central >> 1) \mid (mot\_de\_gauche << 31)) \tag{4.7}$$

et

$$droite \;=\; ((central << 1) \mid (mot\_de\_droite >> 31)) \tag{4.8}$$

Puisque la dimension x de l'image est un multiple de 32 voxels, seul l'accès aux voxels latéraux nécessite la manipulation des données au niveau du bit. Pour les voxels haut, bas, entrant et sortant, aucune manipulation de bits n'est nécessaire ; l'opération ET/OU logique est effectuée directement bit à bit sur les mots de 4 octets.

Prenons l'exemple d'une organisation 2D des données en mémoire. Soient les lignes de voxels $l(x)$ (avec x l'index de la ligne) organisées en mémoire selon un pas régulier p (le *pitch* ou tout simplement la dimension x de l'image) et $c(i, j)$ les coordonnées du voxel central v de l'élément structurant. Les coordonnées de v peuvent s'écrire $c(32k + m, j)$ avec p le pas régulier d'une ligne à l'autre, k l'index du mot dans la ligne $l(j)$ dans lequel le voxel est situé et m sa position dans le mot de 32-bits. Puisque la dimension x de l'image est un multiple de 32 (contrainte mentionnée précédemment, voir 4.5.1), les coordonnées du voxels haut v' sont $c'(32k + m, (j - 1))$.

l(j-1)	mot(1)(j-1)	mot(2)(j-1)	...	mot(k)(j-1)	...	mot(n)(j-1)
l(j)	mot(1(j))	mot(2)(j)	...	mot(k)(j)	...	mot(n)(j)

Pour travailler sur les voxels v et v', on traite les mots qui les contiennent. Une opération ET logique bit-à-bit entre les deux mots permet de récupérer le résultat de l'opération ET logique entre les deux voxels concernés ainsi que sur le reste des voxels contenus dans les des deux mots :

mot(k)(j-1)	01100000	11010010	11111111	10111000
mot(k)(j)	00011010	01011001	10001111	10111100

Une opération logique bit-à-bit entre deux mots traite donc 32-voxels. De même pour les voxels voisins bas, entrant et sortant.

4.5.2 Les stratégies d'implémentation mémoire

Une phase importante du développement est la gestion des allocations mémoires et des transferts de données parmi la hiérarchie complexe des espaces mémoires du GPU. Cette partie détaillera les choix d'implémentation mémoire de l'algorithme. Aussi, puisque le choix de la taille et de la forme des blocs de threads est intimement lié à l'organisation des données en mémoire, ces deux aspects seront traités de front.

Exemple d'implémentation avec la mémoire globale

A l'inverse des générations de cartes 1.x dont la mémoire globale n'est pas cachée, les générations de cartes ≥ 2.x (Fermi et Kepler) sont présentées avec deux niveaux de mémoire cache L1 (max 768 Ko) / L2 (16 à 48 Ko). Pour cela, si l'on choisi de se limiter à l'utilisation strict de la mémoire globale, cela se répercutera par un écart de performances significatif en fonction de la génération de carte choisie : mauvaises performances sur les G80 et GT200 et performances relativement bonnes sur Fermi et Kepler.

Dans un premier temps, dans un soucis de compréhension et pour simplifier le code, nous nous contentons d'implémenter une érosion sur une image 2D (Listing 4.3). Seule la mémoire globale est utilisée et les débordements mémoire sur les bords de l'image ne sont pas traités en supposant que l'espace mémoire alloué est suffisamment grand pour englober les voxels limitrophes sortants. Aussi, les valeurs des voxels hors de l'image sont égales à 0xffffffff pour ne pas interférer avec les valeurs de l'image.

Nous désignerons celle-ci comme une implémentation naïve.

Listing 4.3 – Code CUDA pour l'implémentation naïve d'une érosion sur une image 2D

```
// allocation memoire lineaire
__global__ void erosion2D_gmem( int* o_data, int* i_data,
                                int dimx, int dimy )
{
    int mem_size = dimx*dimy;
    int xindex = blockIdx.x * blockDim.x + threadIdx.x;

    // teste l'espace d'adressage
    if( xindex >= mem_size)
        return;

    // erosion = haut & gauche & central & droite & bas
    o_data[xindex] = i_data[xindex-dimx] & // haut
    ((i_data[xindex-1] << 31) | (i_data[xindex] >> 1)) & // gauche
    i_data[xindex] & // central
    ((i_data[xindex+1] >> 31) | (i_data[xindex] << 1)) & // droite
    i_data[xindex+dimx]; // bas
}
```

Comment optimiser les accès à la mémoire globale ?

Les accès de 4 octets/thread à la mémoire globale peuvent s'effectuer de deux façons différentes.

- Une transaction par thread à un segment mémoire de taille élémentaire. Cela implique un gaspillage de bande passante :
  - pour les générations G80 et GT200, seul $\frac{\text{4 octets utiles}}{\text{32 octets chargés}} = \frac{1}{4}$ des données chargées par thread est exploité et,
  - pour les générations Fermi avec utilisation de la mémoire cache L1, seul $\frac{\text{4 octets utiles}}{\text{128 octets chargés}} = \frac{1}{32}$ des données chargées par thread est exploité.
- Un accès mémoire coalesced. Dans le meilleur scénario, les données sont alignées de façon contiguë en mémoire et le gain est de un accès mémoire coalesced unique contre 16 (resp. 32) transactions sérialisées par demi-warp (resp. warp). Dans le pire, les données ne sont pas alignées en mémoire et le nombre de transactions est 2 accès mémoires coalesced sur des segments mémoires successifs contre 16 (resp. 32) transactions sérialisées par demi-warp (resp. warp).

Le Tableau 4.4 présente les tailles de transaction mémoire (en octets) pour différentes architectures de GPU et l'Annexe B.2 présente les scénarios pour lesquels un accès mémoire coalesced de 4 octets par thread est possible en fonction du type d'architecture GPU.

compute capability	1.0 / 1.1	1.2 / 1.3	> 2.0 sans cache L1	> 2.0 avec cache L1
par thread	32	32	32	128
coalesced par demi-warp	64	64	-	-
coalesced par warp	-	-	4 transactions de 32	128

TABLE 4.4 – Taille du segment de transaction mémoire (en octets) en fonction du type de transaction pour différentes compute capability et des accès de 4 octets par thread.

Les accès mémoires coalesced ont donc deux atouts : un usage efficace de la bande passante mémoire et une réduction du nombre de transactions mémoires. Ils réduisent donc la pression des requêtes mémoires sur la mémoire globale. Il est donc important de prendre ce mécanisme d'accès mémoire en considération pour les applications dont la latence ou le débit mémoire constituent un goulot d'étranglement pour les performances.

Comment organiser les données dans la mémoire globale et quelle forme de bloc de threads choisir ?

Il faut bien comprendre qu'il existe deux grandeurs : les données et les threads. Et pour chaque grandeur, le programmeur peut jouer sur deux paramètres : la quantité et l'organisation

(Tableau 4.5).

	Données	Threads
Quantité	Y-a-t-il suffisamment de ressources mémoire sur le GPU ?	Combien de données doivent être traitées par thread et quelle quantité de threads créer ?
Organisation	Les données ont-elles des dépendances suivant certaines dimensions (1D,2D ou 3D) ?	Si les données ont une dépendance suivant une dimension quelle organisation des blocs de threads (1D,2D ou 3D) possible faciliterait la programmation ?

TABLE 4.5 – Les deux grandeurs à prendre en considération lors du calcul parallèle : les données et les threads et leurs deux paramètres à ajuster par le programmeur : la quantité et l'organisation.

La quantité de données à allouer par thread est un paramètre sensible à aborder et à maîtriser. En effet, tandis qu'à ses prémices une majorité des scientifiques de la communauté du GPGPU prônaient un taux d'*occupancy* élevé pour de meilleures performances, Volkov démontre dans sa présentation à GTC 2010 [125] qu'un faible taux d'*occupancy* peu aboutir à de meilleur performances.

Pour cette raison, en supposant que la quantité d'espace mémoire est suffisante sur le GPU, nous nous contenterons d'étudier les types d'organisation des données en mémoire ainsi que les formes de blocs de threads qui y sont le plus adaptés.

La taille des données Afin d'estimer l'organisation des données la plus appropriée, il est préférable de connaître la taille des données traitées. En effet, les tailles de grille et de bloc étant limitées (Tableau 4.6), ce sont les premières contraintes auxquelles nous sommes confrontés.

	dimensions x * y * z	
compute capability	< 2.0	≥ 2.0
taille de grille maximale	65535 * 65535 * 1	65535 * 65535 * 65535
taille de bloc maximale	512 * 512 * 64	1024 * 1024 * 64

TABLE 4.6 – Tailles maximales de grille et de bloc selon les trois dimensions x, y et z.

Pour une image 2D dont les données sont organisées suivant une seule dimension, avec un thread / pixel, il est possible d'allouer jusqu'à $gridDim.x_{max} \times blockDim.x_{max} = 33553920$ pixels (compute capability < 2.0) ou 67107840 pixels (compute capability ≥ 2.0). Au delà, l'image doit être découpée en régions de taille maximale $gridDim.x_{max} \times blockDim.x_{max}$ et

chaque thread traitera autant de pixels que de régions, soit $(dimx \times dimy) \div (gridDim.x_{max} \times blockDim.x_{max})$. Le plus simple reste toutefois de travailler dans les deux dimensions x et y.

Pour un volume 3D, il est préférable d'éviter une organisation des données suivant une seule dimension. En fait, il est fort probable que la taille de volume ne puisse être contenue entièrement dans ce cas. La dimension z étant moins étendue que les deux dimensions x et y sur les générations de cartes G80 et GT200 ($gridDim.z_{max} \times blockDim.z_{max} = 64$), il peut être plus judicieux de traiter le volume par tranches 2D.

Pour cela, dans un contexte général, que ce soit pour des images 2D ou 3D, l'organisation des données suivant deux dimensions semble, dans un premier temps, le plus approprié.

CUDA offre plusieurs types d'allocation mémoire au programmeur : linéaire (1D), linéaire + pitch (2D, 3D) ou cudaArray (1D, 2D, 3D). Suivant l'organisation des données choisie, une allocation mémoire peut s'avérer plus avantageuse qu'une autre. C'est ce que nous présentons dans la suite.

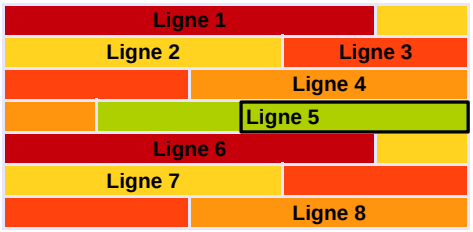
Allocation linéaire avec cudaMalloc Pour permettre des accès coalesced à la mémoire globale, ceux-ci doivent être effectués sur des données contiguës [, alignées et en ordre séquentiel pour les compute capability 1.0/1.1]. De plus, pour réduire le nombre de transactions mémoires coalesced, la dimension x des images doit être un multiple de 16 threads (resp. 32)×32 voxels = 512 voxels (resp. 1024 voxels). Autrement, les accès aux voxels voisins de l'élément structurant ne seront pas alignés. Pour les générations de cartes GT200 et plus, les accès mémoires, bien que coalesced, se retrouvent alors fragmentés sur plusieurs segments mémoires, augmentant le nombre de transactions (Figure 4.8).

Allocation 2D avec cudaMallocPitch L'alignement mémoire peut être ajusté grâce à une fonction CUDA d'allocation mémoire 2D (*cudaMallocPitch*). En substituant l'allocation linéaire de la fonction *cudaMalloc* par la fonction d'allocation 2D *cudaMallocPitch*, la dimension x de l'image est fournie en paramètre et un *pitch* correspondant à l'alignement mémoire GPU le plus proche est rendu par la fonction. La copie de l'image dans cet espace mémoire s'effectue ensuite avec une fonction de copie (*cudaMemcpy2D*) qui impose un alignement mémoire à chaque début de ligne de l'image grâce au *pitch*. Plus précisément, les lignes de l'image sont alignées dans la mémoire globale du GPU selon le *pitch* en ajoutant un *padding* en fin de chaque ligne (Figure 4.9). Les accès à la mémoire globale sont donc parfaitement alignés.

Seulement, ce choix d'organisation des données en mémoire globale présente un inconvénient : la taille minimale du pitch (qui varie selon les GPUs) est de 256 octets (Tableau 4.7). Cela correspond à une dimension x de 256 × 8 = 2048 voxels. Si la dimension x de l'image n'est pas un multiple de 2048, une partie de la quantité mémoire allouée est inutilisée. Cette quantité est d'autant plus importante qu'elle est multipliée par le nombre de lignes (dimension y), voire même la profondeur (dimension z) s'il s'agit d'un volume 3D. Il

	GeForce GTX 285	GeForce GTX 480	Quadro 4000
pitch (octets)	256	512	512

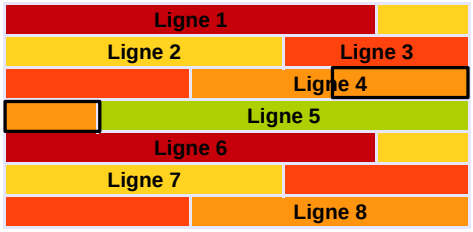
TABLE 4.7 – Taille du pitch de l'allocation mémoire avec cudaMallocPitch pour différents GPUs.



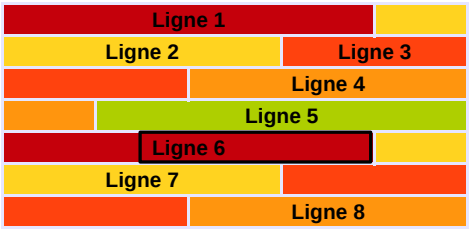
(a) accès coalesced par un demi-warp

demi-warp 1	demi-warp 2
demi-warp 3	demi-warp 4
...	...
...	...
...	...
demi-warp (n-1)	demi-warp n

(b) mapping des warps sur les données



(c) accès non coalesced aux voxels haut de l'élément structurant



(d) accès non coalesced aux voxels bas de l'élément structurant

FIGURE 4.8 – Organisation des données en mémoire lors d'une allocation linéaire avec *cudaMalloc* d'une image de dimension $x = 848$ *voxels* (la dimension x n'est pas un multiple de 32). Le dés-alignement mémoire entraîne des accès non coalesced aux voxels voisins de l'élément structurant.

est donc important de considérer le ratio $r_{mem} = \frac{donnees\ utiles}{donnees\ allouees}$ dans le cas d'utilisation excessive de l'espace mémoire sur les GPUs. C'est un aspect auquel nous avons effectivement dû être confrontés lors du développement sur la GeForce 9500 GT : la binarisation était sensée diminuer la quantité d'espace mémoire allouée. C'était sans compter sur la taille du padding d'une allocation mémoire 2D. La quantité de mémoire embarquée dans les nouvelles cartes est beaucoup plus importante (≈ 1 *Go* minimum). Cependant, bien que les limites de l'allocation mémoire soient repoussées, nous y sommes toujours confrontées lorsque la taille des volumes traités dépassent les 1024^3 voxels.

Comparons maintenant le nombre de transactions mémoires nécessaires pour traiter l'image 2D dans les deux configurations mémoires (Figures 4.8 et 4.9). Dans notre exemple, le schéma d'organisation des données dans la mémoire se répète toutes les 5 lignes. Pour cela, nous

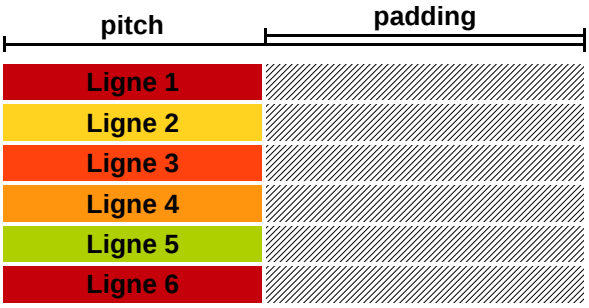
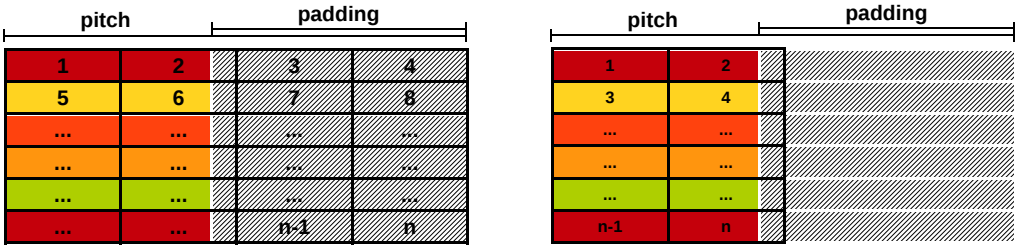


FIGURE 4.9 – Organisation des lignes d'une image 2D lors d'une allocation 2D avec la fonction *cudaMallocPitch*.



(a) Grille 1D. (b) Grille 2D.

FIGURE 4.10 – Allocation de demi-warps pour une grille de treads en 1D (a) et en 2D (b) pour une image de dimension $x = 848$ *voxels* allouée avec cudaMallocPitch et un pitch de 256 octets.

n'effectuons la comparaison que sur les 5 premières lignes. Les déductions suivantes restent cependant valables pour toutes tailles d'images ou de volumes.

- *cudaMalloc* : 8 demi-warps sont nécessaires pour accéder aux 5 premières lignes. 6 transactions mémoire sont nécessaires pour accéder aux voxels voisins de chaque demi-warp (2 pour les voisins hauts, 2 pour les voisins bas, 1 pour le voxel voisin à l'extrémité de droite et 1 pour le voxel voisin à l'extrémité de gauche). Cela fait un total de $8 \times (1 + 6) = 56$ *transactions*.
- *cudaMallocPitch* : chaque ligne nécessite 848 *voxels* $\div$ 32 $\approx$ 27 *threads* pour lire les données en mémoire. Nous arrondissons cette valeur au multiple le plus proche par excès de 16 afin de conserver des accès mémoires coalesced. Bien qu'une partie des données récupérées soit inutile (car appartenant au padding), cet accès coalesced reste avantageux. 2 accès coalesced par 2 demi-warps sont donc effectués pour chaque ligne de l'image. Les données étant parfaitement alignées, les accès aux voisins nécessitent 4 transactions mémoires (1 pour chaque direction : haut, bas, gauche, droite). Cela fait un total de $(5 \times 2) \times (1 + 4) = 50$ *transactions*.

En conséquence, malgré un espace alloué supérieur et des accès mémoires superflus, l'utilisation de la fonction d'allocation 2D *cudaMallocPitch* reste avantageuse par rapport à la fonction d'allocation linéaire *cudaMalloc*.

Suite à l'organisation des données selon l'alignement mémoire du GPU, la taille et de la forme des blocs de threads sont deux caractéristiques du modèle de programmation CUDA qu'il est important de prendre en considération lors d'une implémentation GPU.

Les variables *threadIdx.x*, *threadIdx.y* et *threadIdx.z*, offertent par le modèle de programmation CUDA, permettent une représentation 3D des blocs de threads. Leur but est de faciliter l'adressage des données dans l'espace mémoire : il est, par exemple, plus simple d'exploiter une image quand l'organisation de ses données en mémoire est présentée en 2D. Autrement, le programmeur doit projeter dans son imagination la transformation d'une image 2D en son organisation 1D. Pour cette raison, nous allons étudier l'impact de la forme des blocs de threads dans la grille sur la programmation.

La taille d'un bloc de threads a également un impact sur l'*occupancy* mais cet aspect sera traité ultérieurement (dans la partie résultats).

Blocs de threads 1D Afin de permettre des accès mémoires coalesced, les blocs de threads doivent impérativement disposer d'une dimension x qui soit un multiple de la taille d'un demi-warp, soit 16 threads. Si nous considérons des blocs 1D, ceux-ci seront obligatoirement horizontaux. Verticalement, les accès mémoires ne seraient pas coalesced. L'allocation des threads dans un bloc 1D s'effectue alors de la sorte :

```
dim3 blocks( block_size_x , 1, 1 );
dim3 grids( (mem_size - 1) / blocks.x + 1, 1, 1 );
```

Prenons le cas d'une allocation linéaire avec *cudaMalloc*. Pour que des pertes de performance apparaissent au niveau de warps appartenants à des blocs distincts, il faudrait qu'ils accèdent au même segment mémoire simultanément. Or, il est conseillé que la taille minimale d'un bloc soit de 6 warps, soient 192 threads [125].

Ainsi, en théorie, chaque bloc traite plusieurs lignes de l'image et, à moins de choisir exprès une taille de bloc faible, il est peu probable que ceux-ci traitent, en même temps, des lignes consécutives se chevauchant sur le même segment mémoire.

Malheureusement, en pratique, l'ordonnancement des blocs est inconnu. On pourrait donc imaginer que, tandis qu'un bloc termine le traitement de sa dernière ligne, le bloc suivant débute le traitement de sa première ligne : les accès au même segment mémoire peuvent alors être superposés.

Or, puisque nous ne souhaitons pas appuyer notre étude sur des notions de probabilité, nous nous contenterons donc de retenir que des accès sérialisés au même segment mémoire par des blocs de threads distincts est possible. Nous décrirons le comportement associé comme erratique.

De plus, la taille d'un bloc étant limitée à 512 pour les G80 et GT200 (resp. 1024 pour les Fermi et plus) et la taille de grille 1D à 65535, un adressage 1D limite le nombre de threads alloués à $512 \times 65535 = 33553920$ (resp. 67107840). Ainsi, une limitation de l'espace d'adressage est plus facilement atteignable en exploitant des blocs et une grille 1D. Si nous souhaitons conserver une forme de bloc 1D, cette limitation peut être dépassé en utilisant les autres dimensions de la grille.

Blocs de threads 2D Lorsque les données sont allouées linéairement en mémoire avec la fonction cudaMalloc, l'utilisation de blocs de threads 2D n'apparaît pas comme une option intéressante. En fait, l'utilisation de blocs 2D est même désavantageuse car elle augmente la probabilité que des demi-warps appartenant à des blocs distincts accèdent au même segment mémoire simultanément.

L'utilité de blocs de threads 2D apparaît lors de l'utilisation de la fonction cudaMallocPitch. En effet, nous avons vu que l'utilisation de cette fonction introduit un padding dans l'espace mémoire alloué. Or, sans une représentation en deux dimensions, l'indexation des threads sur la mémoire serait forcément linéaire. Plusieurs threads seraient donc alloués sans être utilisés car ils pointeraient vers les données sans valeur du padding (Figure 4.10a) :

```
dim3 blocks( block_size_x , 1, 1 );
dim3 grids( (pitch * dimy - 1) / blocks.x + 1, 1, 1 );

[...]

// teste l'espace d'adressage
```

```
if( xindex >= pitch * dimy )
    return;
```

Ou bien le programmeur serait appelé à gérer l'indexation 2D :

```
// teste l'espace d'adressage
if( xindex >= pitch * dimy || (xindex % pitch > dimx))
    return;
```

L'introduction d'une représentation en deux dimensions de l'espace des threads alloués permet de mieux cibler les données sur une région d'intérêt 2D (Figure 4.10b). L'allocation des threads est alors modifiée de la sorte :

```
dim3 blocks( block_size_x, block_size_y, 1 );
dim3 grids( (dimx - 1)/blocks.x + 1, (dimy - 1)/blocks.y + 1, 1 );
```

Revenons aux différentes stratégies d'implémentation mémoire possibles. Pour chaque voxel, l'érosion et la dilatation font appel à plus d'un accès mémoire. Ces accès mémoires ont lieu en parallèle sur GPU et leur nombre dépend de l'élément structurant ; dans le cas d'un élément structurant en forme de croix, cela correspond à 5 accès en 2D et 7 accès en 3D.

Considérons les demi-warps 2 et 4 de la Figure 4.10b. Nous remarquons que les voxels voisins bas du demi-warp 2 correspondent au warp 4. De même, les voxels voisins hauts du demi-warp 4 correspondent au demi-warp 2. Donc, pour chaque voxel de l'image, les opérations d'érosion et de dilatation induisent des accès coalesced redondants à la mémoire. Un moyen d'optimiser ces accès redondants est l'utilisation de mémoire cache. Or, pour les cartes GPU de génération G80 et GT200, la mémoire globale ne dispose pas de sa propre mémoire cache. Pour cela, l'utilisation de la mémoire de texture ou de la mémoire partagée pour accompagner la mémoire globale semble davantage appropriée. Intéressons-nous aux avantages qu'apportent chaque type de mémoire pour notre application.

Quelle hiérarchie mémoire choisir pour accélérer les accès redondants à la mémoire globale ?

Sur les GPU de génération G80 et GT200, il est possible d'exploiter la mémoire de texture et/ou la mémoire partagée.

La mémoire de texture est une véritable mémoire cache. Elle ne réduit pas la latence des accès mémoires mais réduit la demande en bande passante à la mémoire globale en soulageant la pression des requêtes mémoires sur celle-ci. Cependant, avec seulement 8 Ko maximum de mémoire de texture embarquée, elle reste inférieure aux 16 Ko de mémoire partagée embarquée par chaque multiprocesseur. Pratiquement, la mémoire de texture présente une palette de paramètres d'initialisation assez large et complexe

	GeForce GTX 285	GeForce GTX 480	Quadro 4000
Bande Passante de la mémoire partagée (Go)	$4 \times 16 \times 0.5$ $\times 1480 \times 30$ $\mathbf{= 1420}$	$4 \times 32 \times 0.5$ $\times 1400 \times 15$ $\mathbf{= 1344}$	$4 \times 32 \times 0.5$ $\times 950 \times 8$ $\mathbf{= 486}$

TABLE 4.8 – Bande passante de la mémoire partagée pour différents GPU.

comparée à la mémoire partagée. Cependant, son utilisation reste avantageuse pour les accès à localité 2D [46], la gestion automatique des cas de bordure à condition d'utiliser des coordonnées normalisées, les opérations d'interpolations rapides à condition que les opérandes soient des flottants.

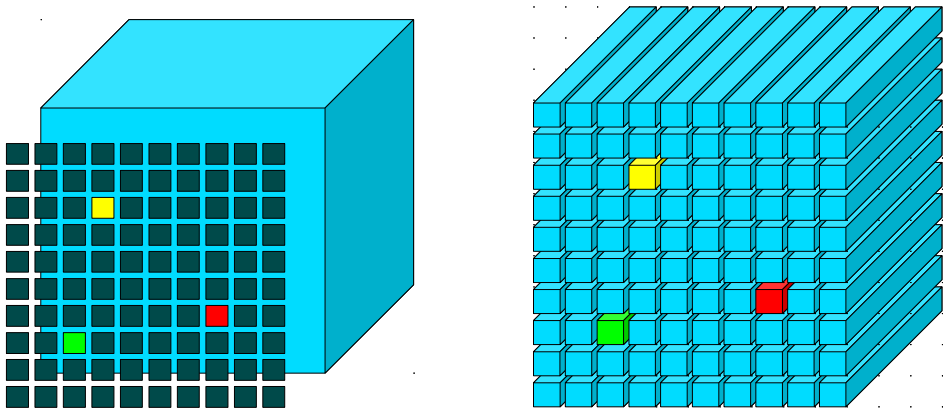
La mémoire partagée Sa latence est du même ordre de grandeur que celle d'un registre et près de x100 fois plus rapide qu'un accès à la mémoire globale. De plus, des accès simultanés par demi-warp (ou warp sur les Fermi et plus) à des bancs distincts de la mémoire partagée sont consumés comme un seul accès. Cependant, des accès simultanés superposés sur le même banc mémoire sont séquentialisés. De plus, la bande passante de la mémoire cache L1 des générations ≥ 2.0 est plus élevée que celle de la texture [85]. Le Tableau 4.8 rapporte la bande passante de la mémoire partagée de différentes architectures grâce à l'équation suivante :

$$\begin{aligned} &4 \text{ octets par banc } \times \text{ \#bancs } \times 0.5 \text{ bancs par coup d'horloge} \\ &\qquad \times \text{ fréquence d'horloge } \times \text{ \#SM} \end{aligned} \tag{4.9}$$

Pour conclure, une première comparaison de performances avec une érosion 2D entre une utilisation de la mémoire partagée et de la mémoire de texture a démontré, dans notre cas d'utilisation, des performances supérieures avec la mémoire partagée. Les débordements mémoires sont toutefois plus simple à gérer avec la mémoire de texture qui les gère automatiquement.

4.5.3 Le cas d'une implémentation en mémoire partagée

Notre approche ressemble à celle de Paulius Micikevicius [78]. Afin de réduire la latence des accès redondants en lecture à la mémoire globale, tous les voxels utilisés plus d'une fois lors des calculs sur un SM sont sauvegardés dans la mémoire partagée. Nous réduisons ainsi la latence des accès en lecture en substituant les accès coûteux à la mémoire globale par des accès moins coûteux à la mémoire partagée. Puisqu'il n'y a pas suffisamment de mémoire partagée pour sauvegarder des données 3D volumineuses, ces dernières sont fragmentées et sauvegardées en mémoire partagée par tranche 2D (suivant le plan (x,y), par exemple). Le volume est alors parcouru suivant la troisième et dernière dimension (respectivement z, par exemple). Ainsi, un thread est assigné à une position (x,y) et calcule successivement le résultat de l'érosion/dilatation pour chaque voxel (x,y) de la dimension z. Tandis que la tranche courante est sauvegardée en mémoire partagée, chaque thread sauvegarde la valeur des voxels (x,y) des tranches précédente et suivante dans des registres.



(a) Découpage du volume par la grille.

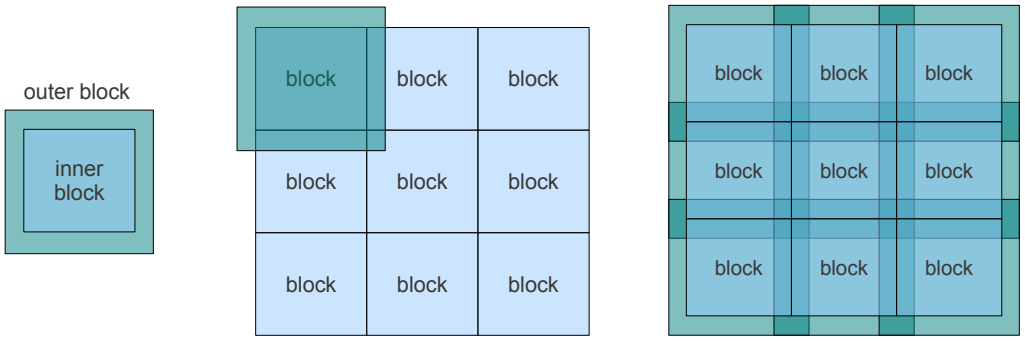
(b) Chaque bloc traite une partition.

FIGURE 4.11 – Etape 1. Découpage du volume avec la grille. Les couleurs de bloc (jaune, rouge, vert) ne sont là que pour illustrer le découpage de la grille. Chaque bloc traite une partition de taille (blockDim.x, blockDim.y) de dimension z = dimz.

Nous pouvons résumé notre implémentation en trois étapes. Premièrement, le volume est découpé en sous-volumes 3D de partition 2D $(\Delta x, \Delta y)$=(blockDim.x, blockDim.y) et de dimension z=dimz (cf. Figure 4.11). Deuxièmement, les effets de bords des partitions doivent être traités. Pour cela, on alloue un espace mémoire partagée de taille (blockDim.x + 2, blockDim.y + 2). Ainsi, chaque thread charge le voxel situé à sa position (threadIdx.x, threadIdx.y) dans la mémoire partagée. Ensuite, deux vecteurs de threads horizontaux (threadIdx.x == 0 et threadIdx.x == 1, par exemple) sauvegarde les deux colonnes gauche et droite des voxels voisins extérieurs. Deux vecteurs de threads verticaux (threadIdx.y == 0 et threadIdx.y == 1, par exemple) sauvegarde les deux colonnes haute et basse des voxels voisins extérieurs (cf. Figure 4.12). Finalement, chaque thread travaille sur une colonne de 32 voxels (un sous-volume du sous-volume) en calculant un mot de 32-bits à chaque incrémentation de z. A chaque passe, le voxel courant (situé en mémoire partagée) est chargé en voxel sortant (dans un registre) et le voxel entrant (situé dans un registre) est chargé comme voxel courant (dans la mémoire partagée). Seul le voxel entrant doit être chargé depuis la mémoire globale (cf. Figure 4.13).

4.5.4 Sommation des voxels d'intérêt sur tout le volume

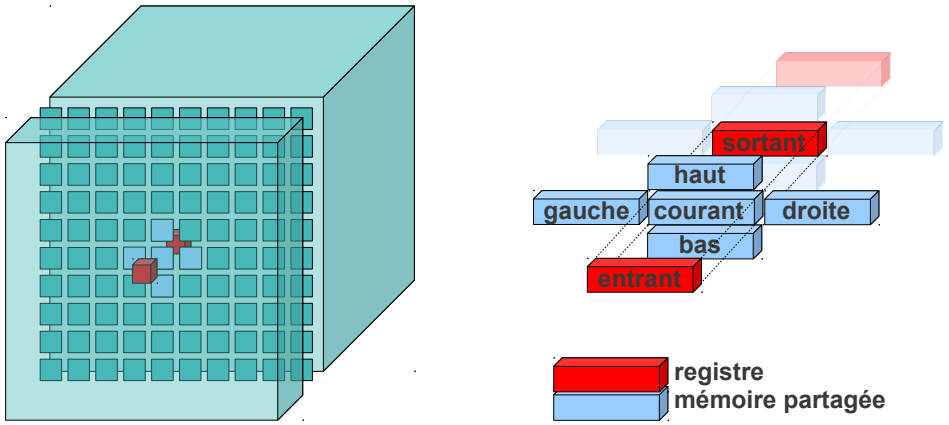
Le comptage des voxels d'intérêts est la dernière étape après chaque ouverture (Figure 4.4b). Un kernel spécifique est dédié à cette dernière étape. La dernière dilatation et le comptage de voxels d'intérêts sont exécutés dans le même kernel afin d'économiser les requêtes vers la mémoire globale : puisque le résultat de la dernière dilatation n'est pas réutilisé ultérieurement, les voxels résultats ne sont pas sauvegardés dans la mémoire globale. Cependant, le résultat de la dilatation doit être sauvegardé temporairement pour effectuer le comptage. Pour cela,



(a) Pour remédier aux effets de bords l'*outer block* permet l'accès aux voxels extérieurs

(b) Recouvrents mémoires dûs aux *outer block*

FIGURE 4.12 – Etape 2. Travail sur les effets de bords. Chaque thread charge le voxel situé à sa position (threadIdx.x, threadIdx.y) dans la mémoire partagée. Ensuite, deux vecteurs de threads horizontaux (threadIdx.x == 0 et threadIdx.x == 1, par exemple) sauvegarde les deux colonnes gauche et droite des voxels voisins extérieurs. Deux vecteurs de threads verticaux (threadIdx.y == 0 et threadIdx.y == 1, par exemple) sauvegarde les deux colonnes haute et basse des voxels voisins extérieurs.



(a) Chaque bloc traite son volume par tranche 2D suivant la dimension z.

(b) Chaque thread de bloc traite des mots contenant 32 voxels.

FIGURE 4.13 – Etape 3. Dans un bloc, chaque thread travaille sur une colonne de 32 voxels (un sous-volume du sous-volume) en calculant un mot de 32-bits à chaque incrémentation de z. A chaque passe, le voxel courant (situé en mémoire partagée) est chargé en voxel sortant (dans un registre) et le voxel entrant (situé dans un registre) est chargé comme voxel courant (dans la mémoire partagée). Seul le voxel entrant doit être chargé depuis la mémoire globale.

un espace de la taille du bloc de threads est alloué dans la mémoire partagée.

Le processus est divisé en sept étapes :

1. Chaque thread calcule le nombre de voxels d'intérêts i dans le mot m de 4 octets : $i = \_popc(m)$.
2. Chaque thread ajoute son résultat i à son emplacement ($threadIdx.x$, $threadIdx.y$) dans la mémoire partagée sh_mem : $sh\_mem(threadIdx.x, threadIdx.y) += i$
3. z est incrémenté et on recommence à la première étape. Une fois que le volume a été entièrement traversé suivant la direction z, on passe à l'étape suivante.
4. Désormais, chaque SM détient à l'emplacement ($threadIdx.x$, $threadIdx.y$) de sa mémoire partagée la somme suivant z des voxels d'intérêts contenus dans le mot de 4 octets à l'emplacement ($blockIdx.x \times blockDim.x + threadIdx.x$, $blockIdx.y \times blockDim.y + threadIdx.y$) du volume.
5. Chaque bloc de threads effectue une réduction sur sa mémoire partagée sh_mem.

```
// unsigned int tid = threadIdx.x + blockDim.x*threadIdx.y;
if (blockDim.x*blockDim.y >= 1024)
    if (tid < 512)
        sh_mem[tid] += sh_mem[tid + 512]; __syncthreads();
if (blockDim.x*blockDim.y >= 512)
    if (tid < 256)
        sh_mem[tid] += sh_mem[tid + 256]; __syncthreads();
if (blockDim.x*blockDim.y >= 256)
    if (tid < 128)
        sh_mem[tid] += sh_mem[tid + 128]; __syncthreads();
if (blockDim.x*blockDim.y >= 128)
    if (tid <  64)
        sh_mem[tid] += sh_mem[tid +  64]; __syncthreads();
if (tid < warp_size)
{
    sh_mem[tid] += sh_mem[tid + 32];
    sh_mem[tid] += sh_mem[tid + 16];
    sh_mem[tid] += sh_mem[tid +  8];
    sh_mem[tid] += sh_mem[tid +  4];
    sh_mem[tid] += sh_mem[tid +  2];
    sh_mem[tid] += sh_mem[tid +  1];
}
```

6. Le résultat de la réduction de chaque bloc de threads, contenu à l'emplacement $sh\_mem[0]$ de la mémoire partagée de chaque bloc, est ensuite additionné dans un compteur final $g\_mem\_compteur$ situé dans la mémoire globale : $atomicAdd(g\_mem\_compteur, sh\_mem[0])$
7. Finalement, quand chaque bloc a sommé le résultat de sa réduction dans le compteur final $g\_mem\_compteur$, la valeur du compteur est transférée sur le CPU.

4.6 Résultats expérimentaux

Nous comparons ici trois implémentations de l'algorithme de granulométrie : deux implémentations CPU multi-threads et une implémentation GPU. Deux de ces implémentations ciblent deux architectures différentes (CPU et GPU) et contiennent les même optimisations algorithmiques présentées à la section 4.2. Puisque le choix d'une implémentation sur 32-bits est une limitation due aux caractéristiques du GPU (Tableau 4.2), la dernière implémentation CPU a été développée sur 64-bits. L'implémentation CPU sur 32-bits a été parallélisée avec OpenMP tandis que la version 64-bits a été implémentée sous la forme d'un plug-in dans le logiciel de traitement et d'analyse d'images ImageJ. Cette dernière version implémentée en JAVA a été développée au SIMaP pour traiter plus efficacement de gros volumes.

Le reste de cette section sera organisé comme suit. D'abord, nous présenterons la station de travail qui a servi de plateforme d'étude. Ensuite, nous analyserons les résultats obtenus sur CPU. Puis, nous effectuerons une étude approfondie visant les variations de performances de l'implémentation GPU en fonction de la taille des volumes traités. En plus d'établir un lien entre les performances et les caractéristiques de l'architecture du GPU, cette étude permettra d'estimer les temps de calculs GPU pour de plus gros volumes. Finalement, nous afficherons le gain GPU par rapport au CPU.

4.6.1 Station de travail

La configuration de la station de travail sur laquelle les mesures ont été effectuées est composée de :

- CPU : Intel Core i7-920 (@2.67GHz) à quatre coeurs
- OS : CentOS release 5.6 (Final)
- Memoire : 11.8 Go
- GPU : GTX285 & GTX480 & Quadro4000 connectées par PCI-E 2.0
- CUDA Toolkit 4.0

4.6.2 Accélération CPU

La Figure 4.14 présente les temps de calcul de l'implémentation CPU avec OpenMP pour différents nombre de threads. Nous observons que le nombre de threads optimal est de 4 (le nombre de cœurs du processeur Intel Core i7 embarqué) ou de 8 (utilisation efficace de l'*HyperThreading*). L'*HyperThreading* permet de multiplier "virtuellement" le nombre de cœurs disponibles en entrelaçant sur le processeur deux flux d'instructions provenant de deux threads différents. Cela a pour but d'optimiser l'utilisation des unités de calcul du processeur. Au vu des résultats quasiment identiques affichés entre la solution sans HyperThreading (4 threads) et la solution avec HyperThreading (8 threads), nous pouvons en déduire que les ressources de

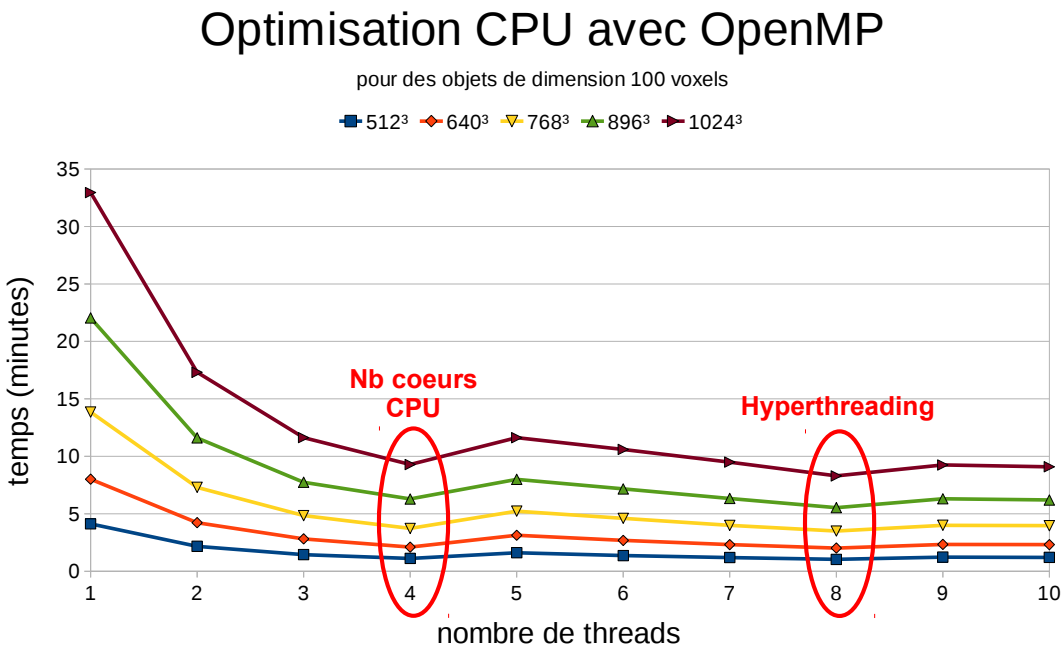


FIGURE 4.14 – Impact du nombre de threads OpenMP sur les performances pour différentes dimensions du volume traité. Nous observons que le nombre de threads optimal est de 4 (le nombre de cœurs du processeur Intel Core i7 embarqué) ou de 8 (activation de l'*HyperThreading*).

calcul sont déjà utilisées de façon très efficace avec 4 threads. Listing 4.4 présente le code CPU parallélisé avec OpenMP ; quelque soit la dimension suivant laquelle la boucle est parallélisée, les résultats sont les mêmes.

Listing 4.4 – Fonction CPU parallelisee avec OpenMP pour le calcul d'une erosion ou d'une dilatation. "granulo_cpu" occupe la place des kernels CUDA dans le pseudo-code de l'algorithme de granulométrie présenté dans Listing 4.1.

```
void granulo_cpu ( unsigned int* input,
                   unsigned int* output,
                   unsigned int dimx,
                   unsigned int dimy,
                   unsigned int dimz,
                   unsigned int (*operation) (unsigned int,
                     unsigned int, unsigned int, unsigned int,
                     unsigned int, unsigned int, unsigned int)
                 )
{
  unsigned int
  current, previous, next, up, down, in, out;
  unsigned int i,j,k;

  #pragma omp parallel private(i,j,k,current,previous,next,
        up,down,in,out) shared(dimx,dimy,dimz,input,output)
  #pragma omp for
  for(k=0; k<dimz; k++)
  {
    // #pragma omp for
    for(j=0; j<dimy; j++)
    {
      // #pragma omp for
      for(i=0; i<dimx; i++)
      {
        current = input[i+j*dimx+k*dimx*dimy];
        if (i==0)
          previous = (current << 1) | (current & 0x00000001);
        else
          previous = (current << 1)
                   | (input[(i-1)+j*dimx+k*dimx*dimy] >> 31);
        if
          (i==dimx-1) next = (current >> 1)
                                | (current & 0x80000000);
        else
          next = (current >> 1)
                   | (input[(i+1)+j*dimx+k*dimx*dimy] << 31);
        if
```

```
            (j==0) up = current;
          else
            up = input[i+(j-1)*dimx+k*dimx*dimy];
          if
            (j==dimy-1) down = current;
          else
            down = input[i+(j+1)*dimx+k*dimx*dimy];
          if
            (k==0) out = current;
          else
            out = input[i+j*dimx+(k-1)*dimx*dimy];
          if
            (k==dimz-1) in = current;
          else
            in = input[i+j*dimx+(k+1)*dimx*dimy];

          output[i+j*dimx+k*dimx*dimy] = (*operation) (current,
                                previous, next, up, down, in, out);
        }
      }
    }
}
```

La Figure 4.15 présente ainsi les performances des différentes implémentations CPU : mono-thread 32-bits, OpenMP 32-bits (8-threads) et le plug-in 64-bits pour ImageJ (8-threads). Cette dernière implémentation s'avère être 3x plus rapide que l'implémentation OpenMP sur 32-bits et 10x plus rapide que l'implémentation mono-thread.

La Figure 4.16 présente le nombre de Cycles CPU Par Pixel. La régularité de cette grandeur démontre bien que le code est adapté à son architecture sous-jacente.

4.6.3 Accélération GPU

La Figure 4.17 présente le temps de calcul GPU pour différentes dimensions du volume de données traitées. Les points de mesures sont espacés de 128 voxels et $128 + 32 = 160$ voxels, soient 128, 256, 384, 512, 640, 768, 896, 1024 et 160, 288, 416, 544, 672, 800, 928, 1056 voxels. Nous nous sommes limités à ces points de mesures à cause du temps grandissant des opérations de calcul sur ces volumes. En plus, une étude approfondie avec davantage de points de mesure est effectuée ultérieurement.

On peut observer que le temps nécessaire pour effectuer une granulométrie sur un volume 1024^3 est inférieur à une minute grâce au GPU. En y regardant bien, on peut aussi apercevoir

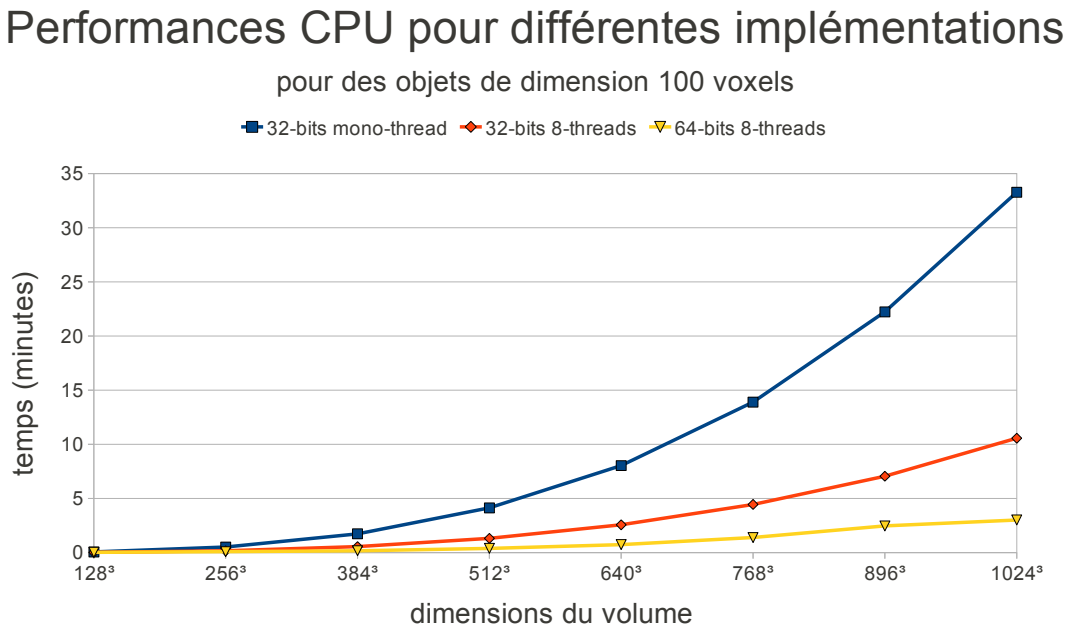


FIGURE 4.15 – Temps de calcul des différentes implémentations CPU sur un volume contenant des objets de 100 voxels. L'implémentation Image étendue sur 64-bits est 3x plus rapide que l'implémentation OpenMP sur 32-bits et 10x plus rapide que l'implémentation mono-thread.

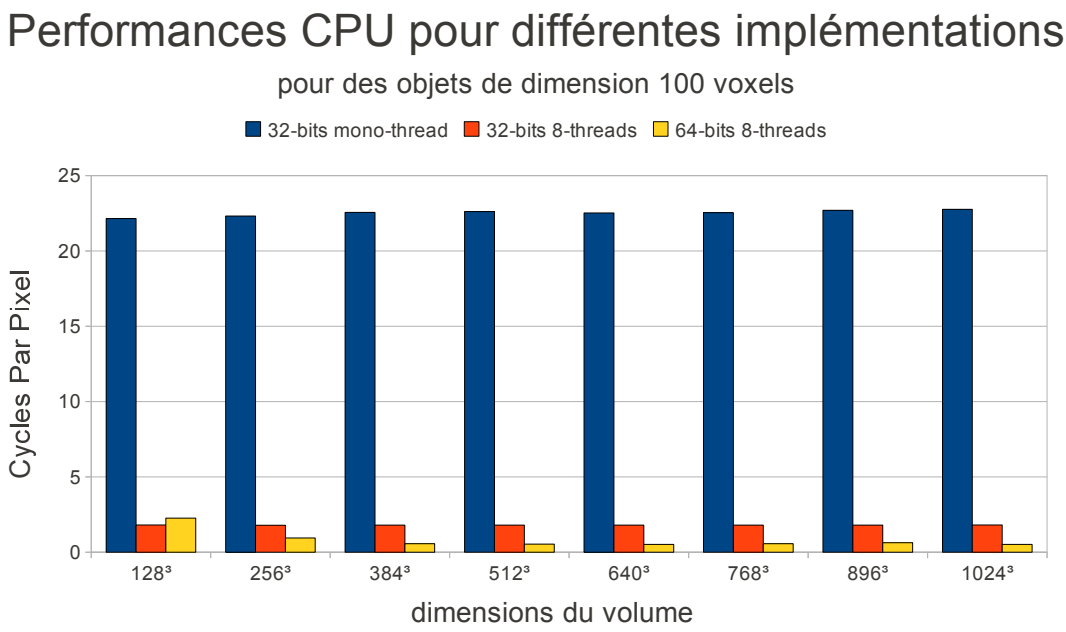
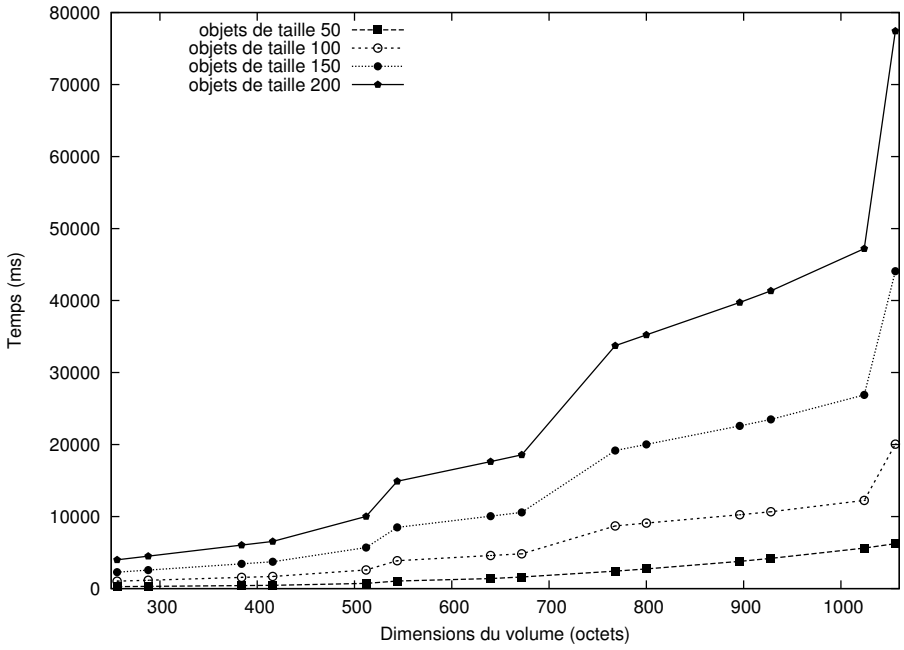
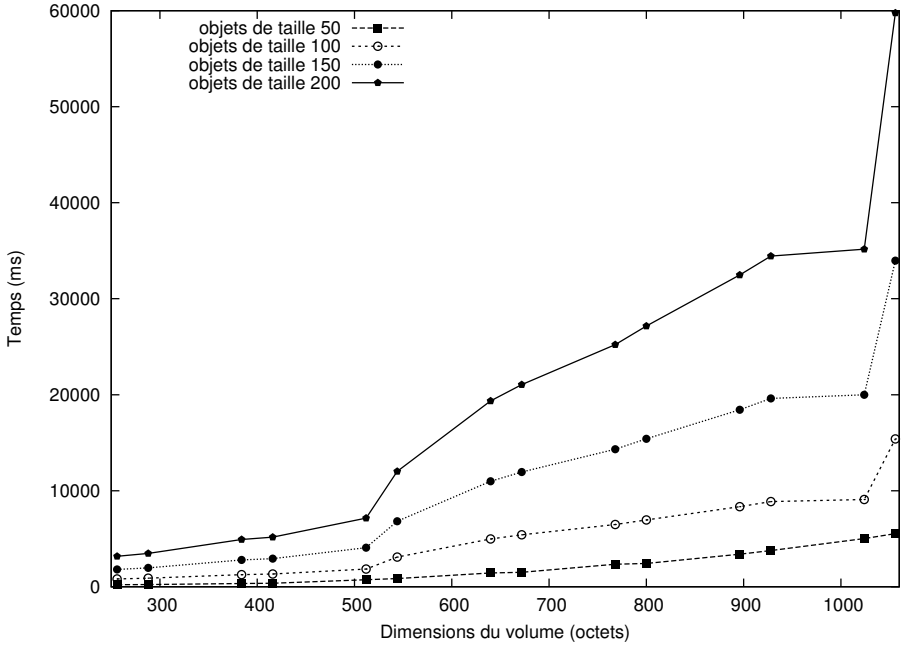


FIGURE 4.16 – La régularité du nombre de Cycles CPU Par Pixel en fonction de la dimension du volume indique que le code exploite de façon efficace l'architecture sous-jacente.



(a) GTX 285



(b) GTX 480

FIGURE 4.17 – Temps de calcul sur deux cartes GPU différentes : la GTX285 et la GTX480 pour différentes dimensions du volume d'entrée. En y regardant bien, on peut apercevoir des paliers avec des sauts périodiques tous les 512 octets. Ce phénomène intriguant est étudié et mis en lumière ultérieurement durant l'étude des variations de performances sur GPU. Sur la GT200, nous observons aussi un saut dans la plage entre 672 et 768 voxels ; cette plage un peu trop large ne nous permet pas de préciser, pour l'instant, la taille de volume à laquelle la variation à lieu.

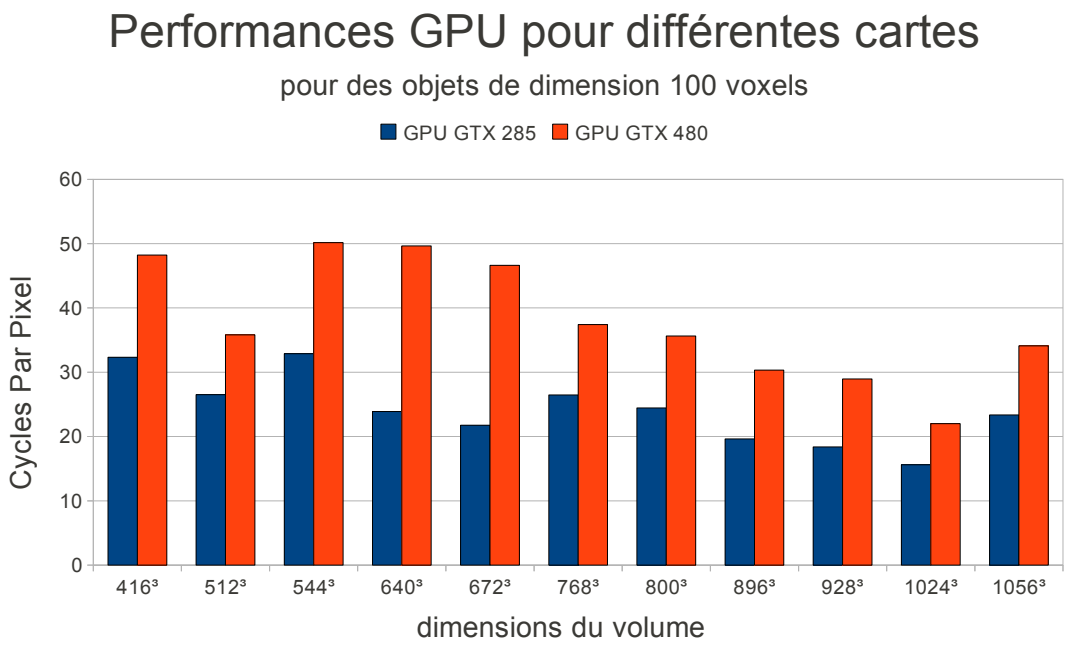


FIGURE 4.18 – L' irrégularité du nombre de Cycles GPU Par Pixel en fonction de la dimension du volume indique que le code exploite de façon inefficace l'architecture sous-jacente.

des paliers avec des sauts périodiques tous les 512 octets. L' irrégularité du nombre de Cycles GPU Par Pixel démontre que le taux d'utilisation matérielle du GPU varie en fonction des dimensions de volume. Ce phénomène intriguant est étudié et mis en lumière ultérieurement durant l'étude des variations de performances sur GPU. Sur la GT200, nous observons aussi un saut dans la plage entre 672 et 768 voxels ; cette plage un peu trop large ne nous permet pas de préciser, pour l'instant, la taille de volume à laquelle la variation à lieu.

4.6.4 Investigation des variations de performances sur GPU

Puisque le souhait du SIMaP est de traiter des volumes plus gros, d'un gabarit d'environs 2048^3 voxels voire même plus, nous allons entamer une étude afin de comprendre les phénomènes de variations des performances de l'implémentation GPU. A terme, cela nous permettra d'estimer les temps de calcul GPU pour de plus gros volumes

L'étude approfondie des résultats sera effectuée sur la GTX285 avec une taille de bloc fixe de 16x16 et un volume de dimensions *variable* x 512 x 512 voxels : une des trois dimensions varie tandis que les deux autres restent fixées à 512. Le pas d'incrémentation est fixé à 32 voxels à cause de la dimension x qui accède à des mots de 32-bits (32 voxels).

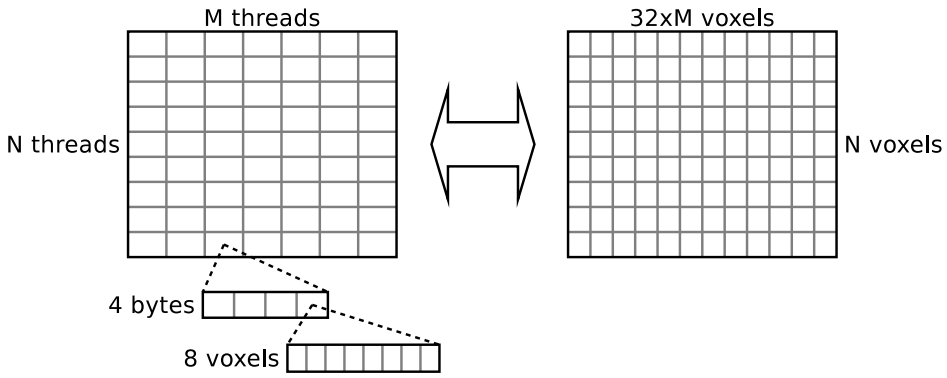


FIGURE 4.19 – Un bloc de dimensions MxN traite une tuile 2D de dimension (32 × M)xN.

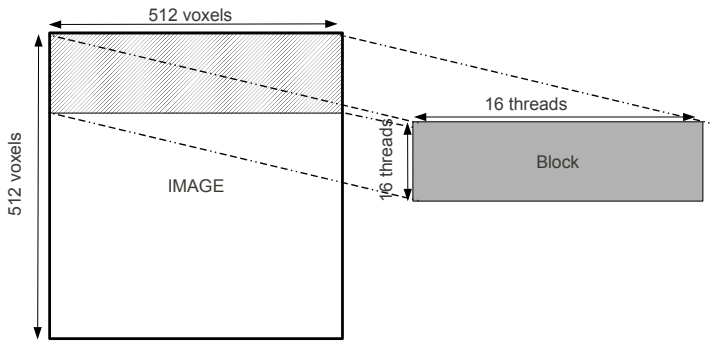


FIGURE 4.20 – Un bloc de threads de taille 16x16 traite 16 lignes de 512 voxels.

Les blocs de threads et la dimension x

Considérons un bloc de threads de taille 16x16, chaque thread traite 4 octets afin de permettre des accès mémoire optimisés (accès coalesced). Aussi, chaque voxel est codé sur 1 bit. Ainsi, pour que tous les threads alloués dans un bloc soient utiles, la dimension x de l'image doit être un multiple de $blockDim.x \times 4\ octets \times 8\ bits = 16 \times 4 \times 8 = 512$ voxels (cf. Figure 4.19). Quand la largeur (dimension x) de l'image est inférieure à 512 voxels, un bloc de threads est alloué pour traiter 16 lignes ($blockDim.y = 16$) du volume d'entrée (cf. Figure 4.20). Le pourcentage de threads utiles augmente proportionnellement à la largeur de l'image jusqu'à atteindre 100% quand la dimension x est égale à 512 voxels. Quand la largeur du volume est inférieure à 512 voxels, une partie des threads créés sont actifs tandis que l'autre partie est inactive. Toutefois, qu'un thread soit actif/inactif n'influence pas la durée d'exécution d'un warp. Puisque le temps d'exécution d'un warp est la somme du temps d'exécution des chemins divergents, tant que les 32 threads d'un warp fonctionnent simultanément ce temps reste constant. Quand la largeur du volume excède les 512 voxels, un bloc supplémentaire est alloué suivant la dimension x. Cela correspond à 32 blocs supplémentaires pour une dimension y fixée à 512 voxels (cf. Figure 4.21). L'allocation des 32 blocs supplémentaires apparaissent clairement sur la Figure 4.22 par un bond temporel de période 512 voxels. Ainsi, il semblerait

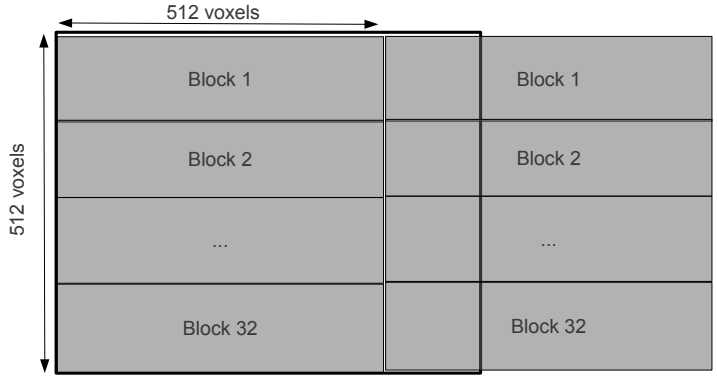


FIGURE 4.21 – 32 blocs supplémentaires sont alloués à chaque fois qu'un nouveau cap de 512 voxels est dépassé.

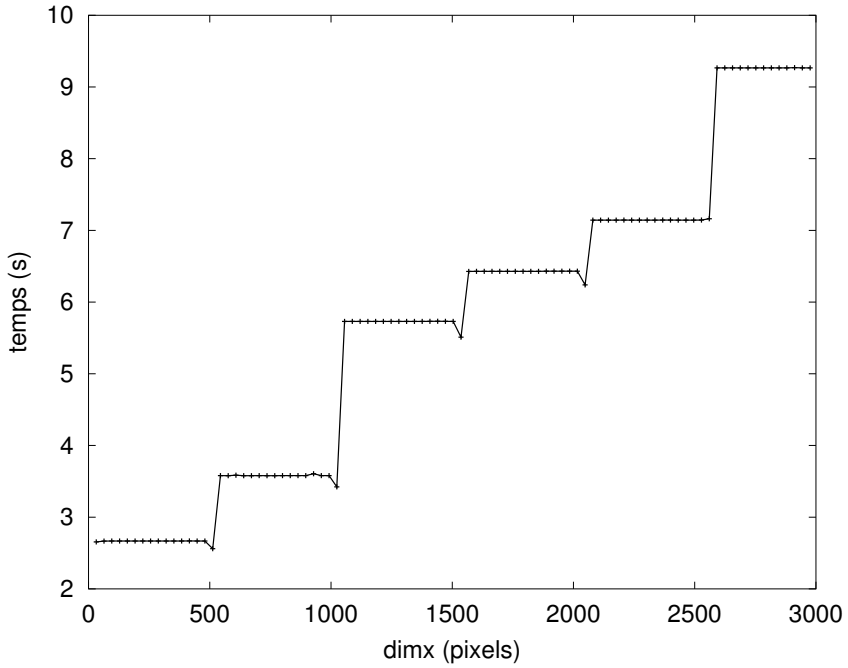


FIGURE 4.22 – Temps d'exécution GPU pour différente valeurs de la dimension x. L'allocation de 32 blocs supplémentaires apparait clairement par un bond temporel de période 512 voxels.

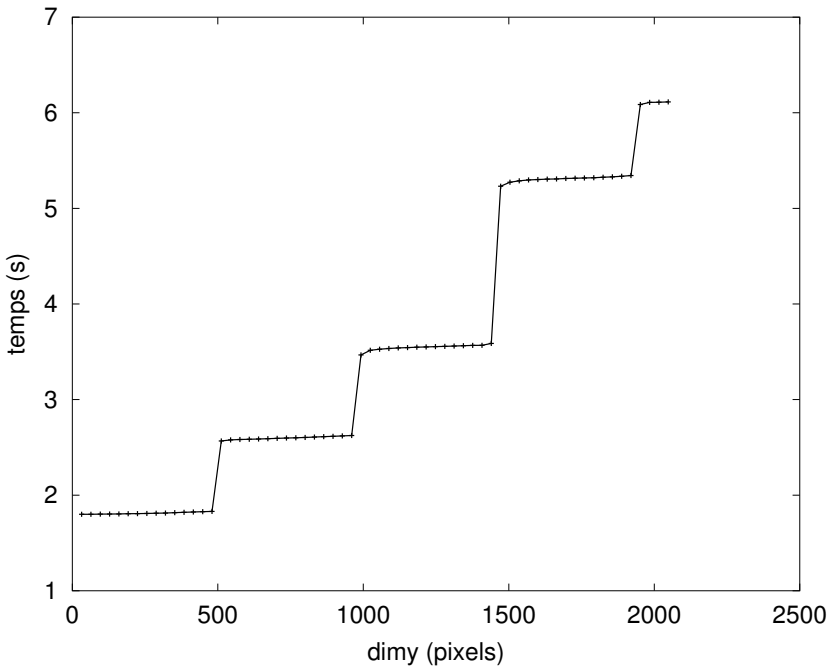


FIGURE 4.23 – Temps d'exécution GPU pour différente valeurs de la dimension y.

que l'allocation de blocs supplémentaires induisent une augmentation du temps de calcul.

Les multiprocesseurs et la dimension y

Dans ce paragraphe, le cas d'étude se limite à un volume de dimension y variable et de dimensions $x, z = 512$ voxels. Donc avec un bloc de threads de taille 16x16, la dimension x est contenue dans un seul bloc (cf. Figure 4.20). En observant la Figure 4.23, nous remarquons un bon temporel tous les 480 voxels. Cependant, avec une taille de bloc $blockDim.y = 16$, pour chaque pas de 32 voxels, $pas \div blockDim.y = 32 \div 16 = 2$ blocs de threads supplémentaires devraient être alloués. Or, l'allocation de ces blocs ne provoque aucune perturbation dans les temps d'exécution de l'application. Ces variations de performances ne sont donc pas liées au nombre de blocs alloués. Un des bénéfices de CUDA est sa scalabilité : il adapte automatiquement le nombre de blocs de threads à procéder sur le nombre de SM que contient le GPU (cf. Figure 4.24). La GTX285 dispose de 30 SM. Quand un bloc est attribué à chaque SM la dimension y est égale à $nb\_SMs \times blockDim.y = 30 \times 16 = 480$ voxels. Quand la dimension y dépasse les 480 voxels, au moins un SM sera occupé par plus d'un bloc (cf. Figure 4.25). C'est donc la charge de travail des multiprocesseurs qui influence le temps d'exécution du GPU. Puisque chaque bloc effectue les mêmes calculs indépendamment des données, leur temps de calcul est identique. Ainsi, le SM qui dispose du nombre de blocs maximal fixe le temps de calcul du GPU : il suffit qu'un SM dispose d'un bloc de threads supplémentaire par

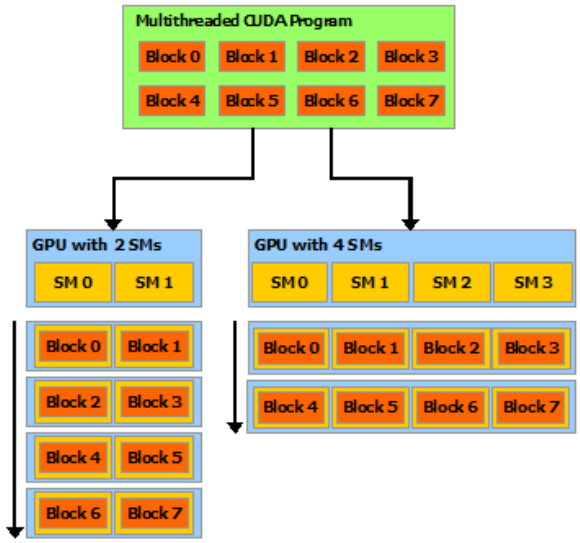


FIGURE 4.24 – Le paradigme de programmation CUDA permet une scalabilité des blocs en fonction du nombre de multiprocesseur embarqués sur la carte GPU.

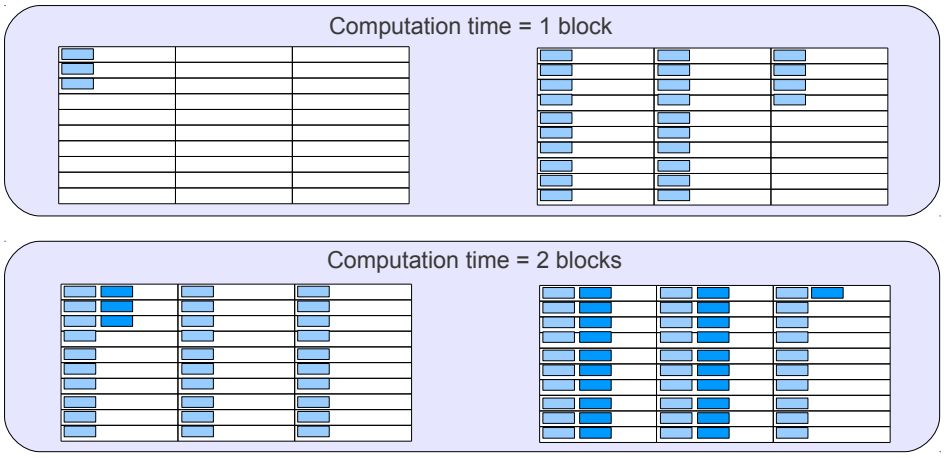


FIGURE 4.25 – Le temps d'exécution d'un GPU dépend de la charge de travail de ses multiprocesseurs. Ce schéma illustre la charge de travail d'un GPU GTX285 constitué de 30 SM. A noter que chaque effectue les même traitement indépendamment des données. Le temps d'exécution d'un bloc est donc constant.

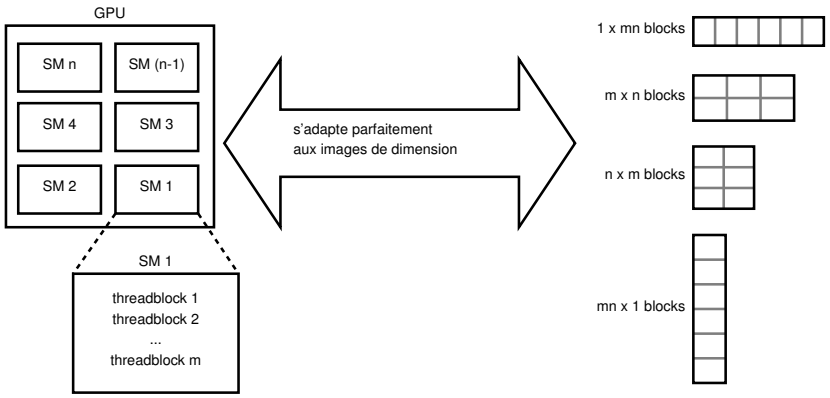


FIGURE 4.26 – Si la carte contient n SMs et chacun peut traiter jusqu'à m blocs simultanément (m est fonction de l'occupancy du programme), à charge maximal de chaque SM, mxn blocs sont ordonnancés simultanément sur la carte.

	compute capability	
	1.x / 2.x	3.x
Le nombre maximal de blocs résidents par multiprocesseur	8	16

TABLE 4.9 – Quantité de blocs de threads que peut traiter un SM simultanément.

rapport aux autres pour que le temps de calcul du GPU soit prolongé de la durée d'exécution d'un bloc de threads. Cela peut être résumé en une équation par :

$$t_{GPU} = (max_{i \in SMs}\ \#blocs(i)) \times t_{bloc}$$

. Les ressources du GPU sont donc exploitées le plus efficacement quand la charge de travail des SM est équi-répartie. Pour cela, le nombre de bloc à procéder doit être un multiple du nombre de SM que contient le GPU.

Pour résumer, les traitements sont organisés suivant deux dimensions :
- spatiale : les données sont découpées en blocs, eux-mêmes répartis sur les SM
- temporelle : si le nombre de blocs dépasse le nombre de SM, les blocs excédentaires sont "empilés" de façon homogènes sur les SM et leur exécution est successive.

L'occupancy et les dimensions x,y

Un bond plus important que les précédents est observé pour les valeurs de x : $dimx = 1024$, $dimx = 2560$ mais aussi pour les valeurs de y : $dimy = 1440$ voxels. L'explication la plus plausible pour expliquer ce bond est le temps de chargement de nouveaux blocs par le *warp*

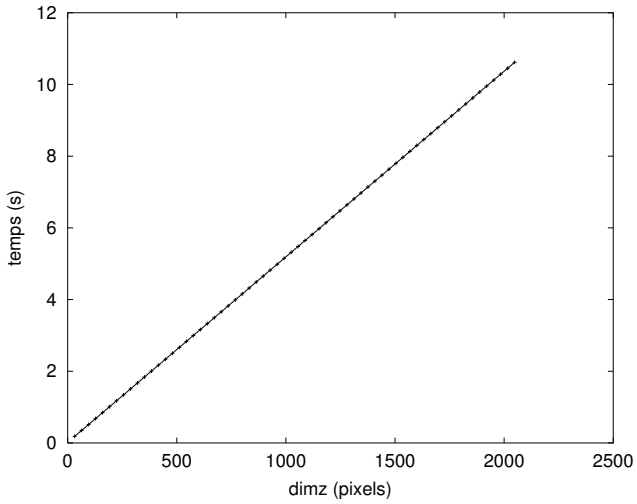


FIGURE 4.27 – Temps d'exécution GPU pour différente valeurs de la dimension z.

scheduler. En effet, bien que chaque SM puisse traiter plusieurs blocs à la fois, il existe une quantité maximale qu'il peut traiter simultanément (cf. Figure 4.26). Cette quantité peut être limitée de deux façons : premièrement, par les normes constructeurs (cf. Tableau 4.9) et deuxièmement, par l'*occupancy* (3.5.3). Pour la GTX285, notre implémentation de l'application de granulométrie limite le taux d'occupation maximal à 3 blocs simultanément par SM. Ainsi, une mise en équation ressemblerait à

$$\#blocs = ceil(dimx \div (blockDim.x \times 32)) \times ceil(dimy \div blockDim.y)$$

$$\#blocs\_per\_SM\_max = ceil(\frac{\#blocs}{\#SM})$$

$$t_{GPU} = \#blocs\_per\_SM\_max \times t_{bloc} + floor(\frac{\#blocs}{occupancy \times \#SM} \times t_{occupancy}$$

Les threads et la dimension z

Quand les dimensions x et y sont fixées, la Figure 4.27 montre que le temps d'exécution des calculs GPU varie proportionnellement aux variations de la dimension z. En fait, cela s'explique par la boucle de calcul suivant la dimension z : à chaque itération, une tranche du volume est traitée, les mêmes calculs sont répétés.

Modèle analytique

La Figure 4.28 récapitule les diverses causes responsables des variations de performances de l'implémentation GPU de l'application de granulométrie.

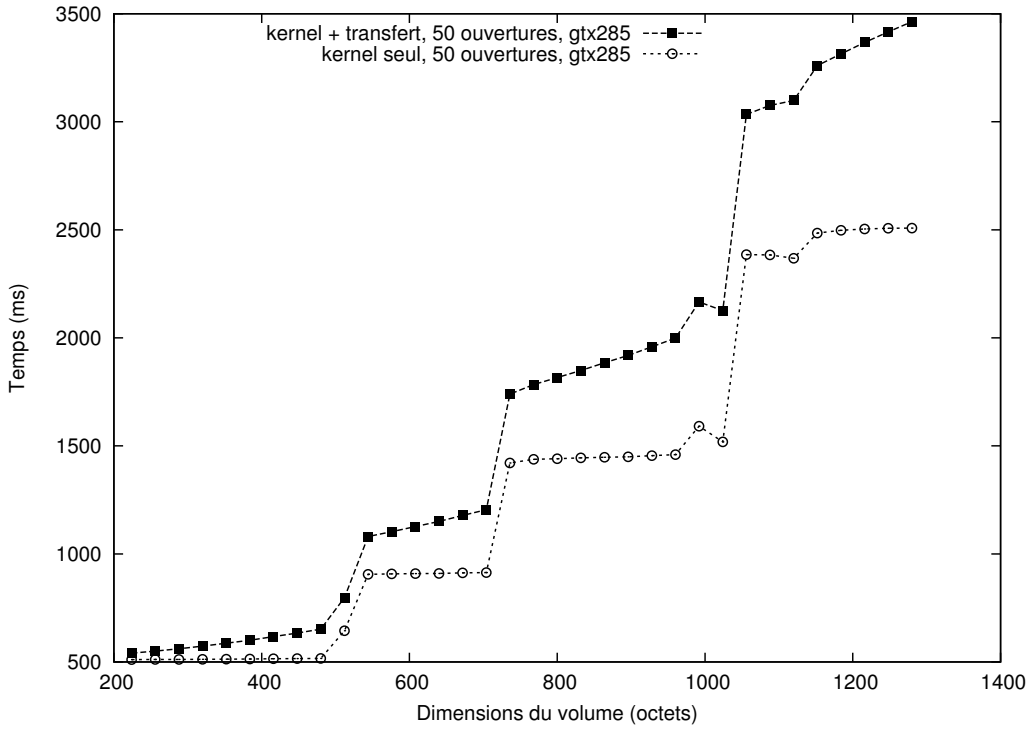


FIGURE 4.28 – Application de granulométrie implémentée sur GPU avec une taille de volume de dimension z fixée à 128. Les deux autres dimensions x et y varient avec un pas de 32 voxels. On observe des bons temporels pour les valeurs 480,960 (augmentation du #_blocs_per_SM à cause de dimy) 512,1024 (augmentation du #_blocs_per_SM à cause de dimx) et 704 (#_blocs_per_SM dépasse l'occupancy).

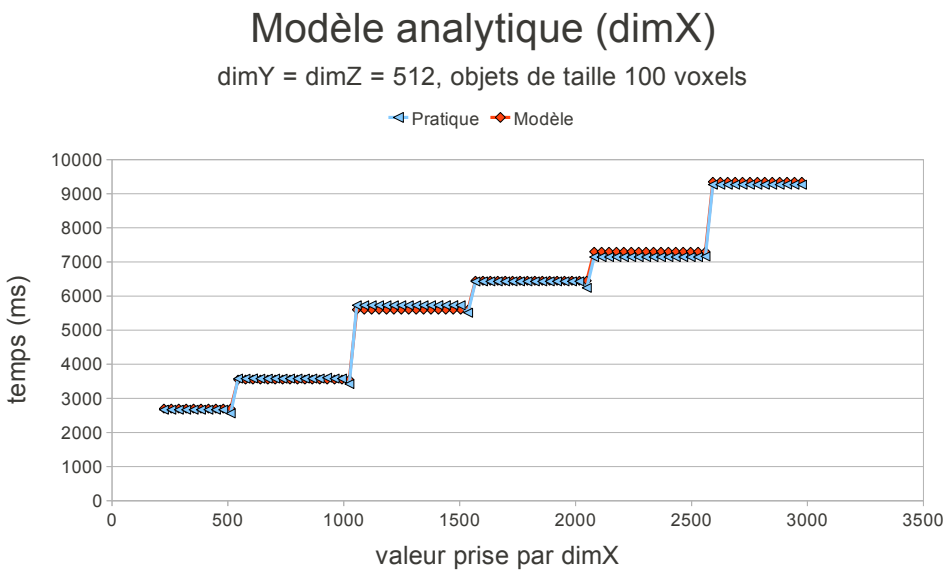


FIGURE 4.29 – Le modèle analytique, faisant varier dimx en figeant dimy et dimz, se superpose quasi-parfaitement aux courbes obtenues par mesure de performances.

A partir des analyses précédentes, il est désormais possible d'établir un modèle analytique décrivant les variations de performance du GPU pour l'application de granulométrie implémentée :

$$t_{total} = t_{GPU} + t_{transfert}$$

$$t_{transfert} = (dimx \times dimy \times dimz) \div debit\_PCI - E$$

$$t_{GPU} = (\#blocs\_per\_SM\_max \times t_{bloc} + floor(\#blocs \div (occupancy \times \#MP)) \times t_{occupancy}) \times dimz$$

Avec :

$$\#blocs = ceil(dimx \div (blockDim.x \times 32)) \times ceil(dimy \div blockDim.y)$$

$$\#blocs\_per\_MP\_max = ceil(\#blocs \div \#MP)$$

Et :

- $dimx$, $dimy$ et $dimz$: dimensions du volume traité
- $\#MP$ et $debit\_PCI - E$: caractéristiques du GPU
- $blockDim.x$ et $blockDim.y$, $occupancy$: choisis par le programmeur
- t_{bloc} et $t_{occupancy}$: déterminés de façon empirique (calibrage sur de petits volumes)

D'ailleurs, les Figures 4.29, 4.30 et 4.31 démontrent bien l'exactitude de ce modèle analytique théorique comparé aux mesures de performances pratique.

4.6.5 Gain du GPU sur le CPU

La Figure 4.32 montre le gain obtenu par l'implémentation GPU par rapport à l'implémentation CPU la plus performante (la version 64-bits) pour différentes architectures de cartes.

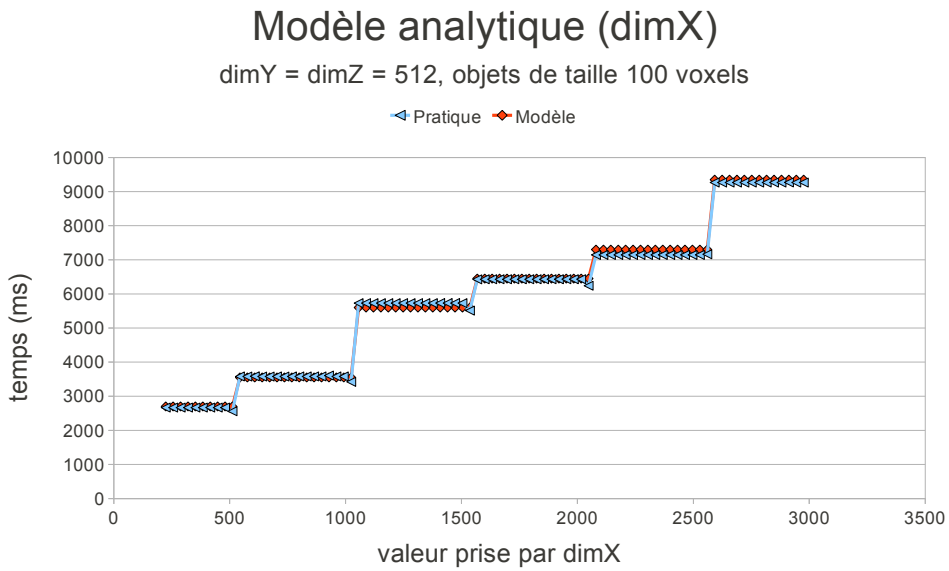


FIGURE 4.30 – Le modèle analytique, faisant varier dimy en figeant dimx et dimz, se superpose quasi-parfaitement aux courbes obtenues par mesure de performances.

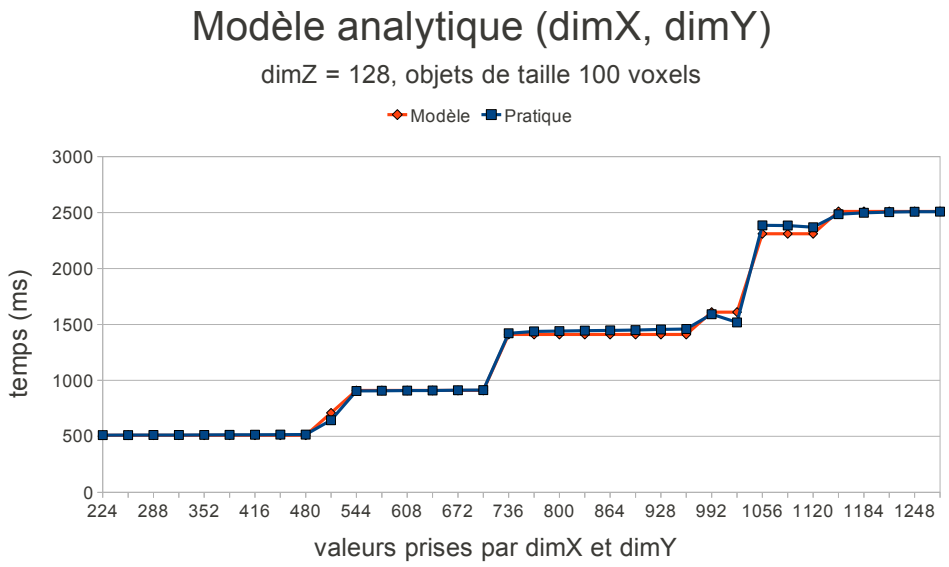


FIGURE 4.31 – Le modèle analytique, faisant varier dimx et dimy en figeant dimz, se superpose quasi-parfaitement aux courbes obtenues par mesure de performances.

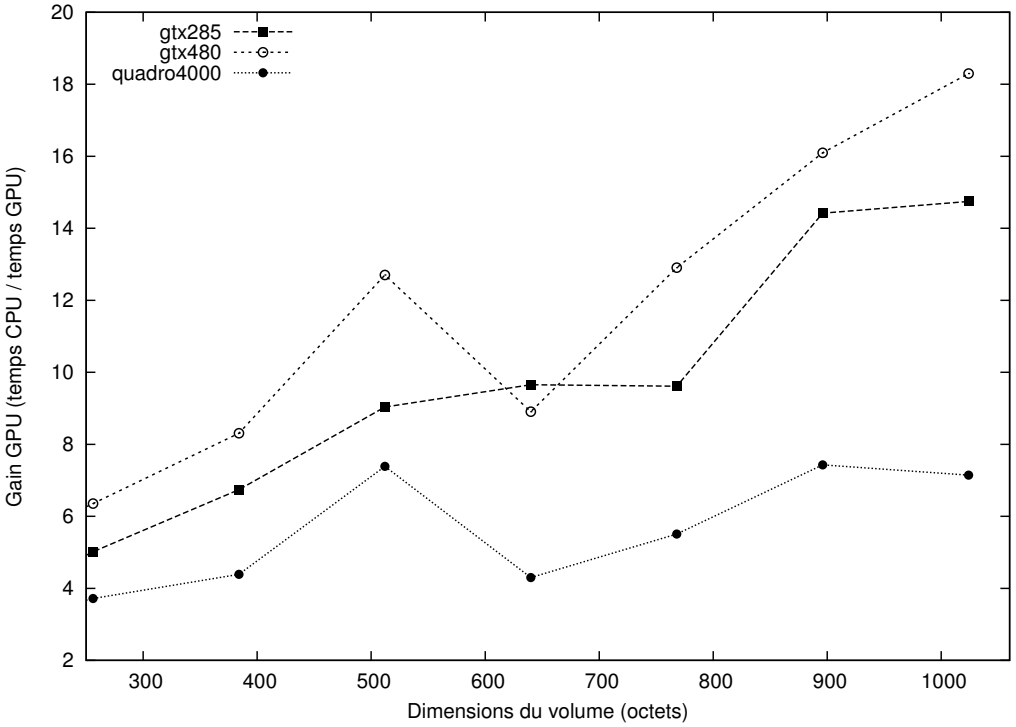


FIGURE 4.32 – Gain du GPU sur le CPU (4 cœurs,8 threads) sur différentes dimensions du volume traité et contenant un objet de 100 voxels maximum (50 ouvertures nécessaires). Les lignes entre les points ne sont pas très pertinentes vu l'écart entre chaque point de mesure. Cependant, ils nous permettent d'observer plus facilement une pseudo allure de la courbe. Nous retrouvons ici des variations du gain liées aux variations de performances sur GPU.

La Quadro n'est pas aussi compétitive que les cartes GeForce car sa fréquence d'horloge est réduite de moitié pour une meilleure fiabilité des résultats de calcul. Nous retrouvons ici aussi les variation du gain lié aux variations de performances sur GPU.

4.7 Conclusion

L'implémentation de l'algorithme de granulométrie sur GPU fut une bonne expérience. Elle nous a permis de saisir la complexité de la programmation sur GPU. L'étude des variations de performances de l'algorithme nous a également permis d'établir le lien entre l'architecture des GPUs et le modèle de programmation CUDA.

4.7.1 Gains en performances

Au final, l'étape de binarisation de l'algorithme de granulométrie optimisé a considérablement accéléré le temps d'exécution. La solution CPU présente déjà un gain en performances relativement important. Tandis que les solutions dont disposait précédemment le SIMaP nécessitaient plusieurs heures de calcul, une granulométrie sur un volume de taille 1024 x 1024 x 1024 voxels nécessite désormais quelques minutes sur CPU et quelques ms sur GPU.

A partir de l'étude des variations de performances de l'algorithme optimisé et implémenté sur GPU, nous pouvons affirmer que le GPU est une solution sereine pour l'accélération de volumes encore plus volumineux. Ses performances bien que irrégulières sont prévisibles.

La seule limitation matérielle à laquelle nous pourrions encore être confronté aujourd'hui avec de tels volumes est la capacité mémoire des cartes GPU à notre disposition. Cependant, les dernières cartes contiennent facilement plusieurs Go et cela ne devrait pas poser de problème que d'allouer 3 espaces mémoires de 2048^3 binarisés, soit $3 \times (2048 \times 2048 \times 2048)/8 \sim 1$ Go.

4.7.2 Retour d'expérience sur le portage d'algorithmes sur GPU

Concernant le portage d'applications sur GPU, il ne faut pas perdre de vue que le but premier du GPU est d'accompagner le CPU en tant qu'accélérateur. Lors du portage d'une application sur GPU, il faut donc considérer le système global CPU-GPU. Dans un premier temps, il est sage de distinguer les parties du code qui seront plus efficaces sur CPU de celles qui le seront davantage sur GPU. Il est aussi important d'estimer à l'avance le gain GPU souhaité et le temps que nous nous accordons pour atteindre ce facteur. Ensuite, on peut se lancer dans le portage d'un algorithme sur GPU.

Par expérience, une programmation naïve du GPU apportera une première phase de résultat avec un gain mineur. Une première optimisation de ce code peut décupler le gain. S'attarder davantage sur l'optimisation n'apportera généralement qu'un gain légèrement supérieur.

4.7.3 Degré d'optimisation de l'algorithme CPU

La parallélisation de la version CPU de l'algorithme de granulométrie aurait pu être plus poussée en utilisant les extensions SIMD des jeux d'instructions tels que SSE ou AVX. Cependant, ces instructions SIMD augmenteraient la taille des opérandes traités. Cela aurait pour résultat de réduire la quantité de volumes qui pourrait être traités. En effet, nous nous limitons déjà à des volumes dont la dimension x doit être un multiple de 32 voxels (ou même

64 voxels avec l'implémentation sur 64-bits). Cependant, traiter des mots de 128-bits (SSE) ou 256-bits (AVX) limiterait le traitement à des images de dimension x multiple de 128 voxels ou 256 voxels respectivement. Une solution serait alors d'ajouter un *padding* (des bits de compensations) à chaque fin de ligne pour obtenir un multiple de 128 ou 256. Aussi, même si la plupart des CPU embarqués dans les ordinateurs tous publiques supportent désormais les instructions SSE, ce n'est pas le cas pour AVX. La question pourrait alors se transposer à "pourquoi optimiser avec les instructions SSE et pas AVX" ?

4.7.4 Vers des gains plus audacieux ?

Dans le cadre d'étude défini au Chapitre 2, nous avons précisé que cet algorithme sera utilisé suite à l'acquisition par tomographie d'une série de volumes. Ainsi, malgré le fait que notre implémentation soit très rapide, il pourrait être intéressant d'accélérer les traitements d'avantage.

Suite à cette première expérience du GPGPU, le potentiel de telles technologies est indiscutable. Dans cette poursuite effrénée du gain en performances, il est dans la continuité de s'intéresser aux solutions multi-GPU et plus généralement aux clusters multi-nœuds de calcul intensif. On parle alors d'applications avec parallélisme à gros grain.

Ainsi, la première question que se pose un programmeur à la recherche de performances est : est-il possible de combiner la puissance de calcul de plusieurs cartes GPUs afin d'améliorer encore les performances ? Nous nous sommes donc penchés sur l'utilité d'une telle implémentation. Une solution multi-GPU est une architecture parallèle à mémoire distribuée (MIMD). Il en découle deux types de parallélisme propices au multi-GPU :

le parallélisme de données : la même tâche est exécutée sur des données différentes ; peut également être interprété comme du parallélisme de tâches SPMD (Single Program Multiple Data).

le parallélisme de tâches MPMD (Multiple Programs Multiple Data) : des tâches différentes sont exécutées sur différents éléments de calcul.

Nous sommes donc confrontés aux deux scénarios suivants : accélérer l'algorithme de granulométrie en répartissant les "tâches" d'érosion et de dilatation sur plusieurs GPUs ou bien lancer plusieurs instances de l'application sur des volumes différents.

Étant donné les fortes dépendances de données entre les "tâches" d'érosion et de dilatation, leur volume conséquent et les temps de traitement déjà très rapides pour un volume, il est plus sage d'envisager une solution avec parallélisme de données. Dans ce dernier scénario, des tests comparant une exécution simple de l'application de granulométrie sur GPU à l'exécution simultanée sur différents GPU ont montré que les temps de calcul GPU sont inchangés : il n'y a pas de contention mémoire due à la lecture simultanée de volumes 1024^3 voxels.

Deuxième partie

Parallélisme à gros grain : déploiement d'applications de traitement du signal et de l'image sur cluster multi-GPU

Chapitre 5

Modèles de programmation parallèle

Sommaire

5.1 Les types de parallélisme 93

5.1.1 Parallélisme de données 94

5.1.2 Parallélisme de tâches 94

5.2 Les types d'architectures parallèles 96

5.3 Les modèles de programmation pour architectures parallèles 97

5.3.1 Les modèles de programmation pour architectures hétérogènes CPU-GPU 99

5.4 Conclusion 103

Dans ce chapitre, nous commencerons par présenter les différentes formes d'expression du parallélisme. Nous présenterons ensuite les classifications permettant de distinguer les types d'architecture parallèle existantes. Finalement, nous étudierons les modèles de programmation existants pour l'exploitation efficace des architectures parallèles.

5.1 Les types de parallélisme

A la fin du chapitre précédent, nous avons rapidement abordé deux types de parallélisme : le parallélisme de données et le parallélisme de tâches. Dans cette section, nous détaillerons davantage ces deux types de parallélisme.

5.1.1 Parallélisme de données

Ce parallélisme peut être exprimé sur deux types d'architecture : les architectures *Single Instruction, Multiple Data* (SIMD) et les architectures *Multiple Instruction, Multiple Data* (MIMD). Le parallélisme de données réalisé par les architectures SIMD consiste à appliquer la même série d'instructions sur des données différentes. Ce parallélisme à grain fin ayant lieu au sein même d'un processeur, la taille des données traitées est relativement petite : 256-bits maximum avec les instructions SIMD AVX d'Intel. A noter que le Xeon Phi, anciennement Intel Many Integrated Core (MIC) et successeur du projet Larrabee, permet des instructions SIMD sur 512-bits.

Le parallélisme de données peut également être exprimé sur les architectures MIMD par un certain parallélisme de tâches : le *task-farm parallelism*, un parallélisme *Single Program, Multiple Data* (SPMD). Les unités de calculs sont indépendantes et, cependant, elles traitent le même programme. La différence avec le parallélisme SIMD se situe à deux niveaux :

- Les unités de calcul, distinctes, exécutent le même programme simultanément mais elles ne se situent pas forcément au même point du programme et elles n'effectuent pas forcément les mêmes instructions (cela dépend des conditions de branchement).
- La taille des données traitées par un programme est quasiment illimitée en théorie (limitée par les ressources de la mémoire RAM en pratique).

5.1.2 Parallélisme de tâches

Sur architecture MIMD, on peut distinguer deux grands types de parallélisme de tâches.

Parallélisme synchrone

Le premier modèle de parallélisme, hérité de la programmation séquentielle des architectures Von Neumann, est le parallélisme *fork-join* séquentiel : lancement, synchronisation(s), fusion. Ce modèle synchrone permet d'assurer un déroulement séquentiel avec des échéances de début et de fin de tâches maîtrisés (Figure 5.1).

L'approche *task-farm parallelism* précédemment décrite est très similaire à cette approche. Ce sont les dépendances entre les tâches qui les différencient. En effet, grâce à l'absence de dépendances entre les tâches exécutées en parallèle, l'approche *task-farm parallelism* est asynchrone.

Ce parallélisme est utilisé dans les schémas de type MapReduce : les données trop volumineuses pour être traitées sur un seul élément de calcul sont découpées en un grain plus fin puis sont réparties sur plusieurs éléments de calcul ; une tâche globale est alors exécutée

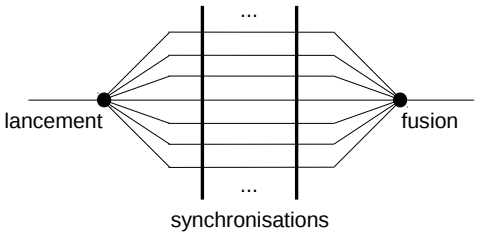


FIGURE 5.1 – Parallélisme synchrone.

sur tout le système. Par exemple, le portage de l'application de Paulius Micikevicius [78] sur plateforme multi-GPU [89].

Il faut toutefois s'assurer, dans ce cadre de travail, que l'application mono-GPU est facilement déployable sur plusieurs GPUs. Cela est parfois rendu impossible à cause de la combinaison de trois facteurs :

- une forte dépendance entre les données : impossibilité d'établir des sets de données distincts suffisamment indépendants pour être lancés sur différents GPUs
- des transferts inter-GPU coûteux en temps à cause de la taille et/ou de la fréquence des échanges
- une programmation complexe et dissuasive

Parallélisme asynchrone

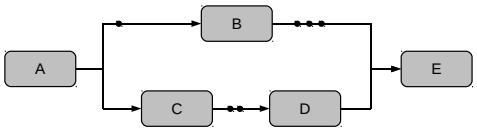
Le deuxième modèle de parallélisme est basé sur le modèle de calcul flux de données. Le modèle *Kahn Process Network* (KPN) est le modèle flux de données le plus générique. C'est un modèle déterministe où la notion de temps n'est pas prise en considération, seule la causalité est importante. L'application est composée de modules indépendants exécutés simultanément et communiquant en s'envoyant des messages dans des files d'attentes *First In, First Out* (FIFO) unidirectionnelles infinies.

La description KPN repose sur trois éléments :

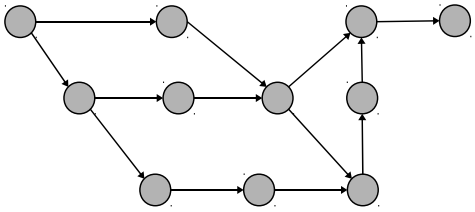
- Acteur : Un acteur représente une unité de calcul (une fonction, par exemple) de l'application. Il peut être producteur, consommateur ou les deux à la fois.
- Arc : Un arc connecte deux acteurs.
- Jeton de données : Un jeton de données est l'élément de donnée atomique échangé entre les acteurs.

En résumé, il existe au moins un acteur-producteur qui produit un/des jeton(s) de données communiqué(s) à au moins un acteur-consommateur par l'arc qui lie les deux acteurs (Figure 5.2a).

Le modèle *Data Flow Process Network* (DPN) [67] ajoute au modèle KPN des contraintes en associant aux acteurs des règles de déclenchement (Firing rules). Ces règles de déclenchement



(a) Le modèle de calcul Kahn Process Network (KPN). Les points sur les arcs correspondent aux données (jetons) qui circulent entre les acteurs.



(b) Expression des tâches d'une application sous la forme d'un *Direct Acyclic Graph* (DAG).

FIGURE 5.2 – Parallélisme asynchrone.

précisent, pour chaque acteur, les conditions d'exécution. Pour aller plus loin, le modèle *Synchronous Data Flow* (SDF) [66] restreint l'expressivité en associant pour chaque arc trois nombres entiers représentant le nombre de jetons produits, le nombre de jetons consommés et le nombre de jetons présents sur l'arc à l'initialisation. Ce modèle apporte une prédictibilité du comportement du système global et permet ainsi d'éviter les *deadlock*, d'effectuer un ordonnancement statique (avant exécution) et de borner la consommation des ressources mémoire. Le modèle SDF peut être restreint à une représentation de type *Direct Acyclic Graph* (DAG) afin d'empêcher les phénomènes de boucle : un chemin passe par un acteur une seule fois. La représentation en forme de DAG exprime visuellement les relations de hiérarchie entre les tâches en fonction des tâches antérieurs qu'elles nécessitent. Elle est souvent utilisée pour représenter une application à mapper sur une architecture (Figure 5.2b).

Ce parallélisme, qui n'est plus synchronisé sur le flot d'instructions mais sur le flot des données, est dit asynchrone. Il peut être réalisé par différents langages, comme Cilk [3], NESL [16], KAAPI [44] ou encore JADE [98]. Ces langages se différencient en fonction des algorithmes d'ordonnancement ou d'accès à la mémoire qu'ils utilisent. Par exemple, Cilk et KAAPI utilisent des techniques de vol de travail (work-stealing [18]) et NESL un parallélisme imbriqué : les tâches lancent les sous-tâches qu'elles peuvent paralléliser.

5.2 Les types d'architectures parallèles

Dans la section précédente, nous avons présenté les types de parallélisme que l'on peut exprimer. Dans cette section, nous décrirons ce qu'est une machine parallèle, les différents types de machines parallèles existantes et comment les distinguer.

Le premier paramètre pris en compte dans la classification des machines parallèles est la centralisation ou la distribution du contrôle. On distingue les machines vectorielles ou SIMD (Single Instruction Multiple Data) des machines scalaires ou MIMD (Multiple Instruction

Multiple Data). A noter que ces deux catégories ne sont pas mutuellement exclusives. D'ailleurs, les processeurs multi-cœurs actuels sont MIMD avec la possibilité d'effectuer du SIMD sur chaque cœur.

Parmi les architectures parallèles MIMD, on distingue les machines à Mémoire Partagée (MP) des machines à Mémoire Distribuée (MD) selon que les accès en mémoire des processeurs se fassent sur une mémoire commune ou exclusivement sur la mémoire propre à chaque unité de traitement.

Un autre type de catégorisation différencie les machines en fonction du type d'accès à la mémoire. Parmi les machines à MP, il existe deux catégories : les machines dont les accès à la mémoire sont symétriques (UMA pour Uniform Memory Access) et celles à accès non-uniformes (NUMA pour Non Uniform Memory Access). Les machines UMA présentent une mémoire commune reliée à toutes ses unités de calcul par un bus de communications unique. Ces machines sont également connues sous la dénomination *Symmetric Multi-Processor* (SMP). Il arrive aussi qu'un commutateur parfait (crossbar) remplace le bus de communication. Ces architectures peuvent rapidement devenir problématiques quand le nombre de processeurs qui partagent le même canal de communication augmente. Pour cela, cette technique est surtout utilisée pour de petits systèmes (typiquement avec moins de 10 processeurs), comme les processeurs multi-cœurs. Les architectures NUMA disposent également d'une mémoire partagée. Cependant, les temps d'accès sont hétérogènes : différentes topologies, différentes technologies de bus mémoire, différents niveaux de mémoire cache, etc.

Parmi les différents types d'accès mémoire, une dernière catégorie subsiste : les *NO Remote Memory Access* (NORMA), l'équivalent des machines à MD. Lorsque des réseaux d'interconnexion par Ethernet ou InfiniBand viennent composer une architecture MIMD, les accès à la mémoire des processeurs s'effectuent par passage de messages.

5.3 Les modèles de programmation pour architectures parallèles

Dans les deux précédentes sections, nous avons abordé les types de parallélismes existants 5.1 et les types d'architectures parallèles 5.2. Dans cette section, nous aborderons les modèles de programmation.

Il y a quelques années, la tendance était au développement de langages de programmation pour les architectures à Mémoire Partagée : Cilk (1994) [3, 17], Posix Threads (1995) [5, 22], OpenMP (1997) [4, 28], *Threading Building Blocks* (TBB) (2006) [97], Sh (recherche académique) commercialisé sous le nom de RapidMind (2004) ensuite acquis par Intel (2009) et intégré au projet de recherche d'Intel *C for throughput* (Ct) (2007) pour finalement composer *Array Building Blocks* (ArBB) (2010)...

Cette tendance a depuis évolué vers des systèmes plus complexes tels que les architectures hétérogènes. Parmi les nombreuses études dans ce domaine, nous pouvons en citer deux importantes :

Sequoia Sequoia [37] est un langage de programmation basé sur l'expressivité de la hiérarchie mémoire. Une application Sequoia est décomposée suivant une arborescence de tâches paramétrées. L'arbre dispose de nœuds qui créent des threads et de feuilles qui effectuent le calcul. Cette arborescence de tâches doit être mappée sur une arborescence mémoire. Sequoia est bien adapté aux applications "Diviser pour régner" mais son utilisation est moins appropriée quand cela sort de ce cadre. Le modèle de programmation devrait être productif et performant pour exploiter des clusters à mémoire distribuée.

Charm++ Un programme Charm++ [58] est composé de chares (l'équivalent de tâches) distribués sur les processeurs de la machine parallèle. Ces chares sont dynamiquement créés et détruit durant l'exécution et ils utilisent des messages pour communiquer entre eux. Charm++ utilise l'API Offload [63] pour décharger des "work requests" vers les SPE du Cell. L'API Offload gère l'exploitation des accélérateurs Cell, coordonnant les mouvements de données, l'exécution des kernels et leur notification de fin de travail. La granularité de parallélisation de l'application est exprimée au moment de l'implémentation à travers les chares. Hybrid API et Asynchronous API sont deux interfaces utilisées pour la programmation sur GPU [128].

Dans la continuité avec plus ou moins de succès, nous pouvons également citer :

Harmony Diamso et al. [31] présentent un modèle d'exécution et un support exécutif pour des systèmes hétérogènes à plusieurs cœurs. Ils souhaitent étendre ce modèle avec des techniques plus poussées telles que l'ordonnancement de tâches avec l'aide de modèles de prédiction de performances. La gestion des données est également étudiée mais se limite à l'expression bas-niveau d'une liste d'adresses.

Merge Merge [69] est une plateforme pour système hétérogène multi-cœur. Elle procure une librairie haut-niveau, basée sur le modèle de parallélisme de données MapReduce, pour la répartition de fonctions sur multiples architectures en sélectionnant la meilleur implémentation pour l'architecture cible. L'implémentation des applications tests a été effectuée sur une configuration Intel Core 2 Duo CPU et une carte graphique 8-cœurs 32-threads Intel Graphics and Media Accelerator X3000. La seule publication date de 2008.

He et al. [52] présentent Mars, une plateforme similaire à Merge (basée sur le parallélisme

de données MapReduce) appliqué à des GPU Nvidia G80.

5.3.1 Les modèles de programmation pour architectures hétérogènes CPU-GPU

Puisque notre attention se porte davantage sur les architectures CPU-GPU, nous définissons ci-dessous quatre critères permettant de distinguer les approches des différents modèles de programmation sur ces architectures : l'effort d'apprentissage nécessaire, l'abstraction de la couche de communication entre les différents éléments de calcul, la gestion de l'ordonnancement des tâches et l'expression d'un modèle de calcul. Certaines solutions combinent également plusieurs de ces axes d'études.

L'effort d'apprentissage nécessaire

Ce sont en fait les modèles de programmation basés sur des directives de compilations pour annoter le code séquentiel (Chapitre 3).

L'abstraction de la couche communication entre les différents éléments de calcul

Nous pouvons citer parmi eux : SnuCL [61], basée sur OpenCL, abstrait les communications entre nœuds de calcul en produisant l'illusion au programmeur d'un nœud de calcul hétérogène unique et SGPU [87], basé sur des processus MPI, pour la répartition de calculs sur CPU et GPU.

SnuCL L'implémentation d'OpenCL est actuellement limitée aux systèmes hétérogènes et ne permet pas l'exploitation de clusters hybrides. SnuCL est une plateforme open-source qui étend la sémantique originelle d'OpenCL pour l'adapter aux environnements de clusters hétérogènes. Le cluster cible est constitué d'un nœud hôte et de plusieurs nœuds de calcul connectés par Gigabit ou par InfiniBand. Le nœud hôte contient plusieurs cœurs CPU et chaque nœud de calcul est constitué de plusieurs cœurs CPU et de plusieurs GPUs. Pour le programmeur, SnuCL produit l'illusion d'un seul système hétérogène. Un GPU ou un ensemble de cœurs CPU est une unité de calcul décrite avec OpenCL. SnuCL permet à l'application d'exploiter des unités de calcul appartenant aux nœuds de calcul comme s'ils faisaient partie du nœud hôte. Ainsi, avec SnuCL, les applications OpenCL écritent pour un système hétérogène composé de plusieurs unités de calcul peuvent être lancées sur un

cluster hybride sans modifications de code. SnuCL promet de hautes performances et une programmation simplifiée dans un environnement cluster hybride.

SnuCL est un support exécutif accompagné d'un compilateur basé sur le compilateur C d'OpenCL de la plateforme OpenCL SUN-Samsung. Le compilateur de SnuCL supporte actuellement les architectures x86, ARM, PowerPC ainsi que les GPU AMD et Nvidia.

SGPU SGPU est un support exécutif pour programmes MPI (tâches communicantes). Les architectures cibles sont les CPU Intel et les GPU Nvidia. L'idée est d'allouer une partie du travail et des données sur chaque matériel ; le développeur doit préparer le travail en définissant des "split function" et des "merge function". Les premières fonctions servent aux transferts et à l'exécution, les secondes à récupérer les résultats. Ces fonctions sont fournies au runtime SGPU qui gère leur exécution parmi les threads CPU et GPU. Ces threads sont organisés en collection appelée "thread set". En plus des fonctions, le programmeur doit donc fournir un "thread set" CPU et un "thread set" GPU au runtime SGPU. SGPU fourni des fonctions de synchronisation, permet le recouvrement des transferts mémoire avec les calculs sur GPU et possède un estimateur de charge *history based time estimator* permettant d'équilibrer la charge de travail lors de travaux itératifs.

La gestion de l'ordonnancement des tâches

Nous pouvons citer parmi eux : KAAPI [44], similaire à Intel TBB et Cilk avec des dépendances de données, dont l'ordonnancement dynamique est décentralisé grâce à sa méthode de vol de travail et StarPU [9], spécialisé dans l'exploitation des GPU, dont l'ordonnancement est centralisé sur la machine hôte qui réparti les tâches sur les éléments de calcul.

KAAPI KAAPI est un support exécutif implémentant un algorithme de vol de travail pour les applications développées sur une plateforme multi-processeurs / multi-cœurs. Il est basé sur l'interface Athapascan [43] qui apporte des extensions au langage C++ : les deux mots-clés *shared* et *fork* ainsi que la représentation des données dans un espace mémoire unifié nommé *global memory*. Le macro DFG est exprimé grâce à deux types d'objets : les *closures*, qui sont les fonctions rattachées aux tâches, et les *accesses*, qui sont les paramètres des fonctions, *i.e.* les données d'entrées/sorties. L'ordonnancement des tâches est établi au cours de la première itération lors du partitionnement du macro DFG sur les unités de calcul à partir de librairies existantes : METIS/Scotch, DSC/ETF, etc. L'algorithme de vol de travail, implémenté par le biais d'extensions mineures des threads POSIX, prend ensuite le relais pour un ordonnancement dynamique des tâches.

X-KAAPI [45] codé en C, présente un overhead beaucoup moins important que sa première version. Les résultats qu'ils présentent montrent des performances équivalente et même meilleure qu'Intel TBB et Cilk. Différents types de paradigmes peuvent être exprimés à

travers cette plateforme : boucles parallèles indépendantes, le parallélisme de tâches *fork-join* et parallélisme de tâches *Data Flow*. Des travaux sont également en cours pour gérer les plateformes hybrides avec accélérateurs [53, 116].

StarPU StarPU est un support exécutif élaboré pour ordonnancer dynamiquement une pile de tâches sur différents types d'accélérateurs connectés à une machine parallèle avec différentes polices d'ordonnancement. Un logiciel de mémoire virtuelle partagée (SVM pour Shared Virtual Memory) permet de connaître la répartition des données sur l'ensemble du système et donc d'éviter les transferts de données superflus. StarPU est un support exécutif local i.e. il est lancé à partir d'un CPU d'où il gère les autres unités de calcul : processeurs et accélérateurs. Pour une exécution sur cluster de calcul hybride, StarPU doit être combiné avec l'interface MPI.

Plusieurs travaux ont adapté StarPU comme support exécutif. Benkner et al. [15] présentent une extension pour le développement d'applications en pipeline sur architectures hétérogènes. Leur travaux est basée sur un mapping des étapes du pipelines sur les unités de calcul composant l'architecture ; chaque étape pouvant présenter autant d'implémentations (optimisées par des experts) que d'architectures cibles. Un compilateur source-à-source traduit l'application pipelinée en une couche de coordination orientée objet greffée sur StarPU.

Dastgeer [30] présente une extention à SkePU [35]. SkePU est une librairie de templates C++ qui fournit une interface simple et unifiée pour le parallélisme de données sur GPU assisté par les patrons de squelettes. Initialement limité au parallélisme de données, Dastgeer incorpore le parallélisme de tâches (farm) à SkePU, une représentation 2D des types de données (utile au traitement d'images) et finalement l'implémentation d'un support exécutif (StarPU) pour l'ordonnancement dynamique de tâches et l'équilibrage de charge parmi les unités de calcul.

L'expression d'un modèle de calcul

Celui qui est le plus répandu et le mieux adapté est, comme nous l'avons présenté à la Section 5.1, le modèle de calcul *DataFlow Process Networks*.

Sbirlea et al. [103] présentent un flot de conception basé sur la spécification DFG d'une application sur une plateforme hétérogènes (CPU, GPU et FPGA). La distribution des tâches sur les unités de calcul est gérée par leur environnement de développement en se basant sur un paramètre d'affinité fixé par le programmeur. Celui-ci est fournit au moment de la spécification de l'application sous la forme d'un DFG : les tâches sont alors attribuées aux unités de calcul en leur affectant une priorité.

```
<denoise\_tag> :: (denoise @ CPU = 20 , GPU= 10 );
```

```
<reg\_tag> :: (registration @ GPU = 5 , FPGA = 10 );
<seg\_tag> :: (segmentation @ GPU = 12 );
```

Leur implémentation dispose également d'un mécanisme de vol de travail (work-stealing). D'un point de vue pratique, leur solution présente toutefois un inconvénient mineur : le programmeur doit présenter autant de versions de code d'une même tâche que d'architectures possiblement ciblées : CPU, GPU ou FPGA. Pour cela, il faut déjà avoir le temps de développer autant d'implémentations. Surtout que le temps d'implémentation et d'optimisation sur ces différentes architectures peut énormément varier. Leur but est de permettre à des experts dans le domaine de l'imagerie médicale de traiter des données sans connaissance du parallélisme ou des détails de l'implémentation en C, CUDA ou VHDL. Ainsi, il est facilement envisageable que ce premier investissement en temps soit rentabilisé dans le cadre du développement d'applications constituées de tâches indépendantes ré-ordonnançables et ré-exploitables. Une librairie de tâches serait ainsi constituée pour le développement rapide d'applications de traitement d'image.

Aldinucci et al. [7] présentent un processus en deux phases ciblant les architectures hétérogènes multi-cœurs et multi-GPU. La première étape vise à traduire des langages haut-niveau en des Macro DFG. Ces graphes sont ensuite exécutés grâce à un interpréteur parallèle de macro DFG spécialisé dans le lancement de calculs parallèles sur GPUs sans intervention du programmeur. L'allocation mémoire, les mouvements de données entre CPU et GPU et l'ordonnancement des tâches sur GPU sont aussi gérés par l'interpréteur. Cependant, peu d'informations nous sont communiquées concernant la possibilité de transferts MPI entre nœuds de calcul d'un cluster. Aussi, l'exécution de squelettes de flot de données parallèles (pipe et farm) ne sont pas présentés. Il serait également intéressant de si l'implémentation du recouvrement calcul-communication est prise en charge par l'interpréteur : allocation de doubles buffers, création de streams CUDA, lancement des transferts, etc

D'autres modèles de programmation inspirés du modèles de calcul DPN proposent à l'utilisateur une interface graphique. Cette fonctionnalités supplémentaire facilite énormément le travail du programmeur. Parmi eux, nous pouvons citer :

SynDEx SynDEx [48] est un logiciel de CAO générant une implémentation optimisée d'une application, spécifiée avec un hyper-graphe orienté acyclique, sur une architecture, spécifiée avec un graphe orienté, constituée de composants programmables tels que le RISC, le CISC, le processeur DSP et/ou des composants non programmables tels que l'ASIC ou le FPGA. Quelques heuristiques sont utilisées pour générer une implémentation optimisée de l'application, *e.g.* la minimisation du temps d'exécution de l'application. Cela résulte en une distribution de l'application sur l'architecture et d'un ordonnancement statique des calculs et des communications. A noter que le parallélisme potentiel de l'implémentation dépend des spécifications de l'application, i.e. Syndex n'opère pas d'analyse atomique des vertices pour

en extraire du parallélisme. A notre connaissance, il n'est pas possible de ciblé un cluster multi-GPU sous SynDEx.

PREESM PREESM (Parallel and Real-time Embedded Executives Scheduling Method) est un outil open source de prototypage rapide et de génération de code. Il est prioritairement utilisé pour la simulation d'applications de traitement du signal et pour générer du code sur les multi-cœurs des DSP. PREESM est développé à l'Institut d'Électronique et de Télécommunications de Rennes (IETR) en collaboration avec Texas Instruments France à Nice.

Les données fournies à l'outil PREESM sont un graphe d'application, un graphe d'architecture et un scénario *i.e.* un set de paramètres et de contraintes spécifiant les conditions de développement. Le type du graphe d'application est une extension hiérarchique des graphes de Synchronous Dataflow (SDF) nommée Interface-Based hierarchical Synchronous Dataflow (IBSDF). Le graphe d'architecture est nommé System-Level Architecture Model (S-LAM). A partir de ces entrées, PREESM mappe et ordonnance automatiquement le code sur les multiples éléments de calculs et génère le code pour multi-cœur adéquat.

5.4 Conclusion

Au delà des standard de programmation parallèle : Pthreads, OpenMP et MPI, de nombreux modèles de programmation ont été développés dans les années passées. Certains assez bas niveau tels que Cilk et Intel TBB ont également su séduire. D'autres modèles plus généralistes ciblent divers types d'architectures parallèles hétérogènes : Sequoia, Charm++, etc. Leur approche est souvent complémentaire aux standards de programmation. Seulement, il est difficile pour le programmeur de les évaluer à cause de la diversité des critères qui les définissent de manière parfois exclusive.

Dans notre cas d'étude, nous nous intéressons essentiellement aux architectures hybrides CPU-GPU. Afin de distinguer les différentes approches des modèles de programmation développés pour ces architectures cibles, nous avons établis quatre critères principaux qui sont : l'effort d'apprentissage nécessaire, l'abstraction de la couche de communication entre les différents éléments de calcul, la gestion de l'ordonnancement des tâches et l'expression d'un modèle de calcul. Bien que les modèles de programmation basés sur des directives de compilation puissent faciliter considérablement le travail de portage d'algorithmes séquentiels, ajouter du parallélisme à une conception séquentielle de l'application est une démarche d'adaptation qui ne peut en aucun cas être comparée à une démarche de conception parallèle d'une application. Par contre, les approches basées sur les flux de données sont plus appropriés pour exprimer efficacement le parallélisme d'une application puisque l'organisation entre les tâches n'est pas pensée selon un enchaînement temporel mais selon l'inter-dépendances de leurs données. Pour aller plus loin, l'usage d'un modèle de calcul peut augmenter l'efficacité d'implémentation d'une application au prix d'une expressivité du modèle de programmation

qui sera moindre.

Pour conclure, parmi ces modèles, KAAPI et PREESM nous semblent être des approches à regarder davantage dans le détail. Pour le premier, la stratégie d'ordonnancement dynamique décentralisé de ses tâches devrait permettre une scalabilité aisée de l'application sur des systèmes relativement larges. Pour le second, la représentation visuelle d'un graphe de flot de données présente un flot de conception ergonomique qui devrait faciliter l'expression du parallélisme dans la phase de conception et de mapping d'une application sur une architecture parallèle.

Chapitre 6

PACCOlib : une librairie pour cluster multi-GPU

Sommaire

6.1 Notre flot de conception 106
6.1.1 Points d'entrée du programmeur 107
6.1.2 Initialisation 112
6.1.3 A l'exécution 116
6.2 Cas d'étude 117
6.2.1 Expérience de déploiement sur plusieurs noeuds de calcul . . 117
6.2.2 Granulométrie 122
6.2.3 Modèle pour le calcul des cartes de saillance visuelle 124
6.3 Conclusion 129
6.3.1 Discussion autour du parallélisme temporel (ou bien parallélisme de pipeline) 132

Les ordinateurs récents embarquent en plus de leurs CPU multi-processeur multi-coeur au moins une carte graphique permettant un affichage graphique 3D de qualité. Le GPU embarqué sur cette carte est dorénavant capable d'effectuer du calcul scientifique à haute performance pour un effort de développement relativement faible comparé à d'autres accélérateurs tels que le Cell et le FPGA. Suite à la première expérience de portage détaillée dans le Chapitre 4, la question d'un portage multi-GPU de l'algorithme de granulométrie a été étudiée. Après refléxion, il en est ressorti que seules certaines applications trouvent un intérêt à un portage multi-GPU et la granulométrie n'en fait pas partie. Nous aurons toutefois l'occasion de tester les effets d'un portage multi-GPU de l'application de granulométrie.

Dans ce chapitre, nous nous plaçons dans un cadre d'étude bien spécifique qui est le Traitement du Signal et de l'Image (TdSI) en *streaming*. Les applications *streaming* de

TdSI présentent trois caractéristiques favorables au portage multi-GPU : (1) d'abord, le flux de données entrant doit être traité séquentiellement, (2) ensuite, ces données traitées séquentiellement présentent des dépendances qui imposent un certain ordonnancement dans l'exécution des tâches et (3) finalement, ce type d'application peut présenter des contraintes temporelles : un débit sortant proche du temps réel, par exemple.

Toujours dans ce cadre, nous nous pencherons sur l'exploitation de solutions multi-GPU. De telles machines connectées via une interface réseau à bande passante élevée (InfiniBand) sont utilisées dans les systèmes de Calcul Haute Performance (HPC pour High Performance Computing). Le principal défi de ce type de système consiste à utiliser efficacement et de façon optimale les ressources de calcul. Pour cela, en plus du code source développé dans un langage adapté aux éléments de calculs, le programmeur doit maîtriser différentes API nécessaires à l'établissement des communications CPU-CPU et CPU-GPU. En plus, il doit s'assurer du bon cheminement des données entre les éléments de calcul. Cela nécessite une gestion des allocations et des transferts mémoire sur tout le cluster. Aussi, il faut pouvoir ordonnancer les tâches et synchroniser leur exécution en fonction du débit des données potentiellement variable d'un niveau à l'autre.

Afin de simplifier la tâche du programmeur, un modèle de programmation haut niveau abstrayant toutes ces opérations simples en soit mais sujettes à erreurs et complexes à gérer collectivement est nécessaire. Dans ce chapitre, nous présentons un flot de conception permettant une implémentation efficace d'applications *streaming* de TdSI sur un cluster multi-GPU.

Dans la première section, nous présentons notre approche basée sur une spécification DFG de l'application de TdSI. Cette représentation permet une expression naturelle du parallélisme. Dans la deuxième section, nous présentons les résultats expérimentaux de trois cas d'études : un *benchmark* multi-host multi-gpu, une implémentation multi-GPU de l'algorithme de granulométrie et finalement, une application *streaming* de saillance visuelle.

6.1 Notre flot de conception

PACCOlib (Parallel Computation-Communication Overlap library) est une solution basée sur une spécification DFG de l'application. Cette représentation permet un découpage aisé de l'application en tâches. Tandis que le choix de la répartition des tâches sur les éléments de calcul revient à l'utilisateur, notre contribution se situe au niveau de l'abstraction de la couche de programmation nécessaire aux communications entre les éléments de calculs. En d'autres termes, si nous considérons que le portage d'applications sur un cluster multi-GPU est composé de deux parties inter-dépendantes : d'un côté, la programmation des unités de calcul et de l'autre, l'implémentation des communications qui les relient, nous concentrons nos efforts sur une implémentation automatisée de ces communications. En plus, l'utilisateur a la possibilité de choisir entre deux modes de fonctionnement : synchrone ou asynchrone. Dans

le cas d'un fonctionnement synchrone, les communications et les calculs sont séquentialisés. Dans le cas d'un fonctionnement asynchrone, il y a recouvrement entre les calculs et les communications. Ainsi, les temps de transferts non négligeables peuvent être masqués par les temps de calcul. Si nous nous reportons aux critères évoqués au chapitre précédent afin de distinguer les modèles de programmation adaptés aux architectures hybrides CPU-GPU, nous pouvons dire que notre outil répond complètement au deuxième critère : abstraction de la couche de communication.

Concernant la spécification de l'application sous la forme d'un DFG, les acteurs peuvent être répartis sur le même élément de calcul et/ou sur des éléments de calcul différents (CPU ou GPU) ses arcs implémentés sous différents types de canaux de communications (passage de pointeurs, PCI-E, InfiniBand, Ethernet, etc) peuvent véhiculer différents types de données (scalaire, vecteur, matrice, etc). Tandis que les tâches mappées au sein d'un élément de calcul (CPU ou GPU) sont exécutées séquentiellement, les éléments de calcul fonctionnent en parallèle. Ainsi, afin de réguler le trafic et que la cohérence des données soit préservée, nous avons choisi d'introduire une synchronisation globale du système. Les différences de latence entre les données transférées en interne (dans un élément de calcul) et en externe (lors d'un parcours sur plusieurs éléments de calcul) sont traitées grâce au modèle d'implémentation Bulk Synchronous Parallel (BSP) [118] : des sets de sous-DFG, un par élément de calcul, communiquent par une synchronisation globale. Après que toutes les tâches appartenant à un élément de calcul ont terminé leur exécution, le thread qui gère cet élément de calcul attend la terminaison des autres threads (des autres éléments de calcul) à la barrière de synchronisation globale.

Notre flot de conception présente des points d'entrées favorisant l'efficacité de développement d'applications de TdSI. Il fait abstraction de la génération de plusieurs parties de code en automatisant leur production afin d'alléger le travail du programmeur. Celles-ci concernent :

- une allocation mémoire (optimisée) sur CPU et GPU
- l'expression des communications inter-processeurs
- la synchronisation des tâches entre les éléments de calcul

La première sous-section présente les points d'entrées du programmeur pour exprimer la répartition des tâches de l'application de TdSI sur un cluster de calcul. La deuxième sous-section discute du processus d'initialisation du support exécutif. Finalemment, la dernière sous-section présente le processus d'exécution en fonction du mode synchrone ou asynchrone choisi.

6.1.1 Points d'entrée du programmeur

Les deux principaux points d'entrée du programmeur sont le graphe d'application *Data Flow Graph* (DFG) (cf. code 6.1) et le graphe d'architecture (AG pour Architecture Graph) (cf. code 6.2).

Le DFG de l'application est inscrit textuellement dans un fichier texte. Il est composé

d'acteurs représentant les calculs et d'arcs représentant le flot des données. La sémantique d'un DFG est la suivante : un acteur peut être lancé si et uniquement si tous ses jetons d'entrée sont disponibles. Une fois lancé, il consomme toutes ses données d'entrée en exécutant la fonction à laquelle il est associé. A l'issu de son exécution, les données résultats produites sont transmises via les arcs connectés à cet acteur. Un seul type de donnée est attribué à chaque arc et une seule et unique fonction est attribuée à chaque acteur.

Tâchons d'être précis dans la présentation du modèle exprimé par notre outil : puisque la quantité de jetons consommés et produits est fixée, nous employons un sous-modèle de DFG qui est le SDF (Chapitre 5). Le modèle peut être raffiné encore davantage puisque nous limitons (pour l'instant) la quantité de jetons produits et consommés à l'unité. Ainsi notre DFG est en fait représenté sous la forme d'un DAG (Figure 5.2b).

Quant aux données, tous les types sont acceptés. Seulement, s'ils n'existent pas déjà dans la bibliothèque, c'est à l'utilisateur de renseigner leur nature et les paramètres nécessaires à leur allocation lors de l'initialisation.

Listing 6.1 – Exemple de représentation textuelle du DFG exprimant la relation producteur-consommateur d'une application.

```
# nodes
NODE
      type , fctnode
      name , producteur
      kernel , capture , frame_out
      out , frame_out , matrix
      mapping , cpu , 0
END
NODE
      type , fctnode
      name , consommateur
      kernel , display , frame_in
      in , frame_in , matrix
      mapping , cpu , 0
END

# edges
EDGE
      type , matrix
      name , cpu_to_cpu
      connect , producteur , frame_out
      connect , consommateur , frame_in
END
```

La Figure 6.1 présente un exemple de DFG : l'acteur `p` produit des jetons consommés

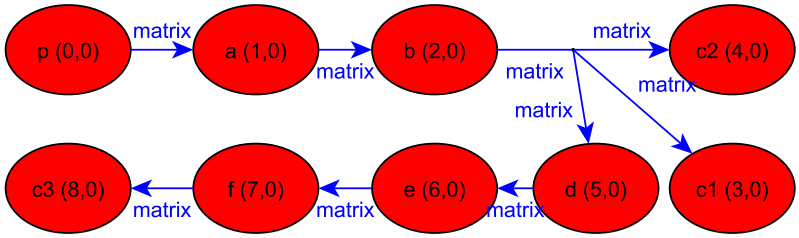


FIGURE 6.1 – Exemple de graphe de flux de données (DFG) utilisé pour représenter une application.

par l'acteur `a` qui produit lui-même des jetons pour l'acteur `b` qui broadcast ses jetons aux acteurs `c1`, `c2`, `d`, et ainsi de suite. Une itération est l'exécuton de tous les acteurs du DFG. Chaque acteur contient une étiquette constituée de son nom et d'un couple contenant son rang d'ordonnancement et sa latence entre le point d'entrée des données dans le DFG et son déclenchement ; nous aborderons ces aspects plus en détail dans la sous-section 6.1.2.

A noter que la décomposition du DFG est manuelle. Nous avons adopté une approche naïve parce que le but de l'outil n'est pas d'aider dans le processus de parallélisation mais de simplifier son déploiement sur cluster. En d'autres termes, le programmeur maîtrise son algorithme et possède déjà une bonne compréhension de ses goulets d'étranglements. Sa tâche se limite alors à découper l'algorithme en tâches de granularité suffisamment fine pour être réparties de façon équilibrée sur les éléments de calcul du cluster ; il décide alors de l'élément de calcul le plus approprié pour chaque tâche. Cependant, rien n'empêche à l'utilisateur de solliciter au préalable des compilateurs paralléliseurs afin de simplifier la parallélisation d'une application.

Listing 6.2 – Exemple de représentation textuelle du graphe d'architecture d'un noeud mono-GPU.

```
# nodes
NODE
      type,cpu
      name,cpu0
      hostname,dhcp-invite5
      inout,ib,ib
      inout,pcie0,pcie
      inout,pcie1,pcie
      inout,pcie2,pcie
      mapping,cpu,0
END

NODE
      type,gpu
      name,gpu0_cpu0
```

```
    hostname,dhcp-invite5
    inout,pcie,pcie
    mapping,cpu,0,gpu,0
END

# edges
EDGE
    type,pcie
    name,e_cpu0_gpu0
    connect,cpu0,pcie0
    connect,gpu0_cpu0,pcie
END
```

Le Graphe d'Archictecture est décrit à l'aide d'une représentation textuelle d'un graphe acyclique. Les noeuds représentent les éléments de calcul et les arcs les canaux de communication les reliant. Chaque arc détermine également la nature du canal de communication qu'il représente. La Figure 6.2 présente le graphe d'architecture d'un cluster de deux ordinateurs composés d'un CPU et de trois GPU chacun. Sur notre cluster, CPU et GPU appartenant à un même ordinateur communiquent par le biais du canal PCI-E tandis que les ordinateurs communiquent par réseau Infiniband.

Le *mapping* d'une application sur l'architecture est spécifié manuellement dans le DFG et l'AG ; il est pris en compte et interprété par l'outil à l'initialisation. La Figure 6.3 montre un exemple de *mapping* possible entre l'application DFG et le graphe d'architecture présentés plus tôt.

Le code compilé et lié grâce à la bibliothèque PACCOlib génère un fichier binaire capable d'effectuer du recouvrement calculs-communications. MPI étant utilisé pour les communicatins inter-CPU, nous utilisons mpirun afin de distribuer puis lancer l'application sur le cluster. Lors de l'exécution, le DFG avec ses annotations de mapping et le Graphe d'Architecture sont analysés. Des transformations graphiques sont réalisés pour permettre

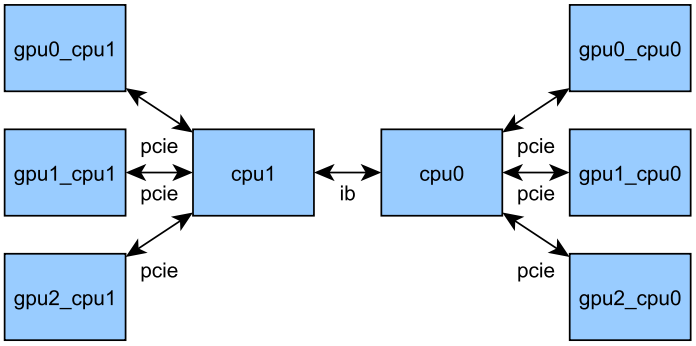


FIGURE 6.2 – Exemple de Graphe d'Architecture.

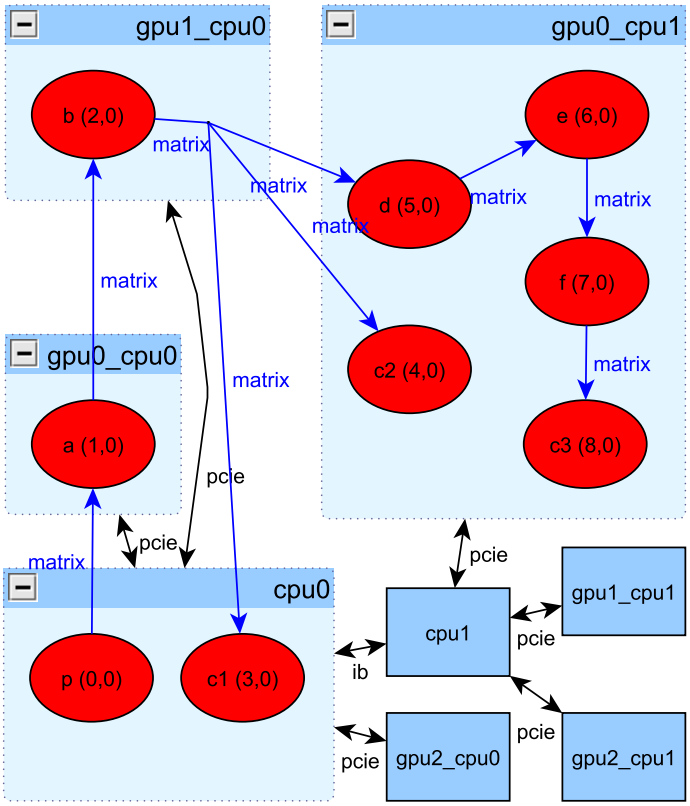


FIGURE 6.3 – Mapping du DFG sur le Graphe d'Architecture.

le recouvrement calculs-communications et pour obtenir une allocation mémoire optimisée. Nous créons ensuite les *threads* qui permettront de gérer les tâches CPU et GPU : déclenchement des acteurs du DFG, transferts mémoires et synchronisations. Ce code doit être recompilé seulement lorsque le concepteur introduit un nouveau type de données (fonctions de résolution) dans le DFG ou associe une nouvelle fonction (classe C++) pour un acteur du DFG.

Pour résumer, le rôle de l'utilisateur est le suivant :

1. fournir une application sous la forme d'un DFG,
2. fournir un graphe d'architecture sous la forme d'un graphe acyclique,
3. fournir une description des classes C++ encapsulant les fonctions associées aux acteurs du DFG,
4. compléter les "fonctions de résolution" qui sont appelées à l'initialisation pour allouer les espaces mémoires.

La Figure 6.4 résume toutes les étapes du processus de conception et d'initialisation. Le concepteur ne doit fournir que les informations contenues dans les rectangles gris foncés.

6.1.2 Initialisation

L'objectif de l'analyse graphique est d'obtenir un Graphe d'Implementation (IG) qui contiendra les informations utilisées par chaque *thread* (CPU ou GPU) pour déterminer le chemin du flux de données, une allocation optimisée des buffers mémoire, l'ordonnancement des fonctions (acteurs) et leur répartition entre les éléments de calcul. Le processus d'initialisation est composé de cinq étapes détaillées dans les paragraphes suivants : l'ordonnacement des acteurs, l'insertion des buffers mémoire entre les acteurs, la taille des buffers, l'optimisation des allocations mémoire et finalement le calcul de la latence pour chaque acteur.

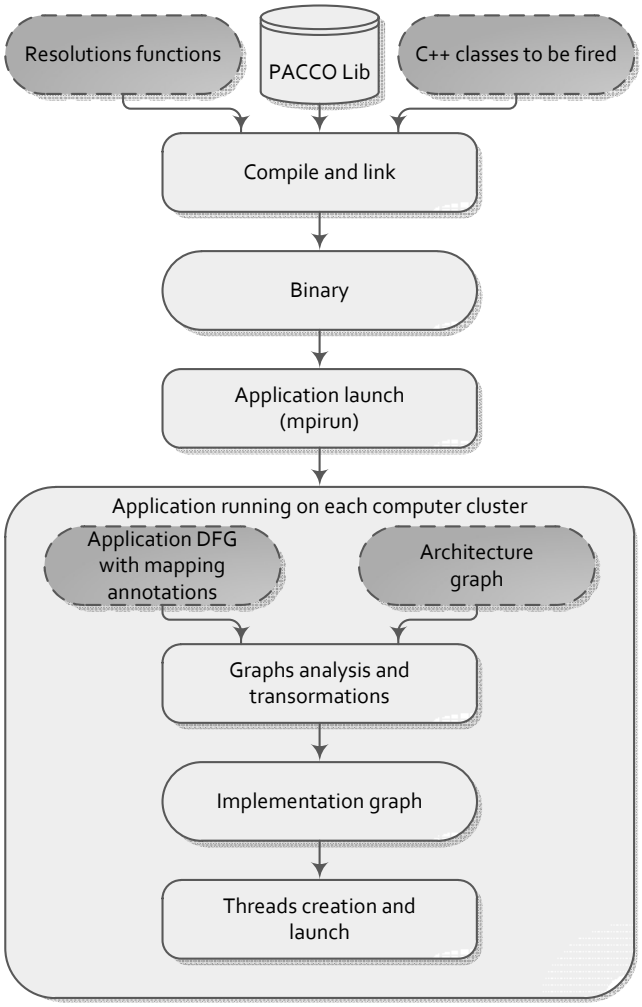


FIGURE 6.4 – Le flot de conception et d'initialisation assisté par PACCOlib. En gris foncé, les informations à fournir par l'utilisateur.

Ordonnancement

La première étape consiste à trouver un ordonnancement *i.e.* l'ordre de déclenchement des acteurs pour chaque itération du DFG. Pour ce faire, un algorithme récursif est utilisé dont le principe est le suivant : un acteur peut être ordonnancé si et seulement si tous ses prédécesseurs le sont, faute de quoi on tente d'ordonnancer ses prédécesseurs. Chaque fois qu'un acteur est ordonnancé, il prend la valeur d'un compteur qui est ensuite incrémenté. Le rang d'un acteur est spécifié dans le premier élément du couple figurant sur son étiquette.

Insertion des buffers mémoire

Le DFG d'une application spécifie les dépendances de données (arcs) entre les fonctions (les acteurs). Du point de vue de la mise en oeuvre, nous avons besoin de buffers pour stocker les données qui sont produites et consommées par les acteurs. Cela signifie que les buffers sont nécessaires entre les acteurs. Ceci est facilement effectué sur le DFG en intercalant des buffers entre les acteurs. Cependant, en mappant le DFG sur le Graphe d'Architecture, des contraintes spécifiques à l'architecture peuvent étendre le chemin du flux de données. Lors d'un mapping des acteurs à l'intérieur d'un élément de calcul, l'insertion de buffers est régulière (entrelacée avec les acteurs). Cependant, lors d'un mapping des acteurs sur différents éléments de calcul, le chemin du flux de données devient plus long et sa longueur peut varier d'une interface à une autre. La Figure 6.5 illustre le résultat après insertion des buffers sur le DFG de l'application (présentée dans la Figure 6.1) mappée sur le Graphe d'Architecture (de la Figure 6.2) suivant le mapping proposé dans la Figure 6.3.

Considérons par exemple le cas des acteurs `b` et `c2` qui ont une relation producteur-consommateur. L'acteur `b` est mappé sur le GPU 1 de CPU 0 alors que `c2` est mappé sur le GPU 0 de CPU 1. Pour aller de GPU 1 sur CPU 0 à GPU 0 sur CPU 1, il faut passer par CPU 0, puis par CPU 1. Ainsi, quatre buffers sont nécessaires : (1) `bn_2` alloué dans la mémoire GDRAM de GPU 1 sur l'ordianteur CPU 0 (2) `bn_9` alloué dans la mémoire RAM de l'ordinateur CPU 0 (3) `bn_10` alloué dans la mémoire RAM de l'ordinateur CPU 1 (4) `bn_11` alloué dans la mémoire GDRAM de GPU 0 sur l'ordinateur CPU 1. Nous remarquons que les données produites par `b` sont également consommées par `c1`. Comme le chemin du flux de données allant de `b` à `c1` est inclus dans le trajet de `b` vers `c2`, aucun autre buffer mémoire n'est nécessaire.

L'algorithme qui insère les buffers se comporte comme suit : un buffer mémoire est inséré sur toutes les sorties des acteurs du DFG. Les buffers insérés sont mappés sur le même élément de calcul que celui auquel les acteurs appartiennent (cas des buffer `bn_0 .. 5` sur la Figure 6.5). Pour chaque buffer inséré, on cherche le chemin physique (dépendant de l'architecture) à emprunter pour atteindre l'acteur cible. Sur le parcours, pour chaque élément de calcul traversé, si aucun buffer n'a déjà été inséré, un nouveau buffer est inséré (cf. les deux chemins empruntés dans l'exemple précédent).

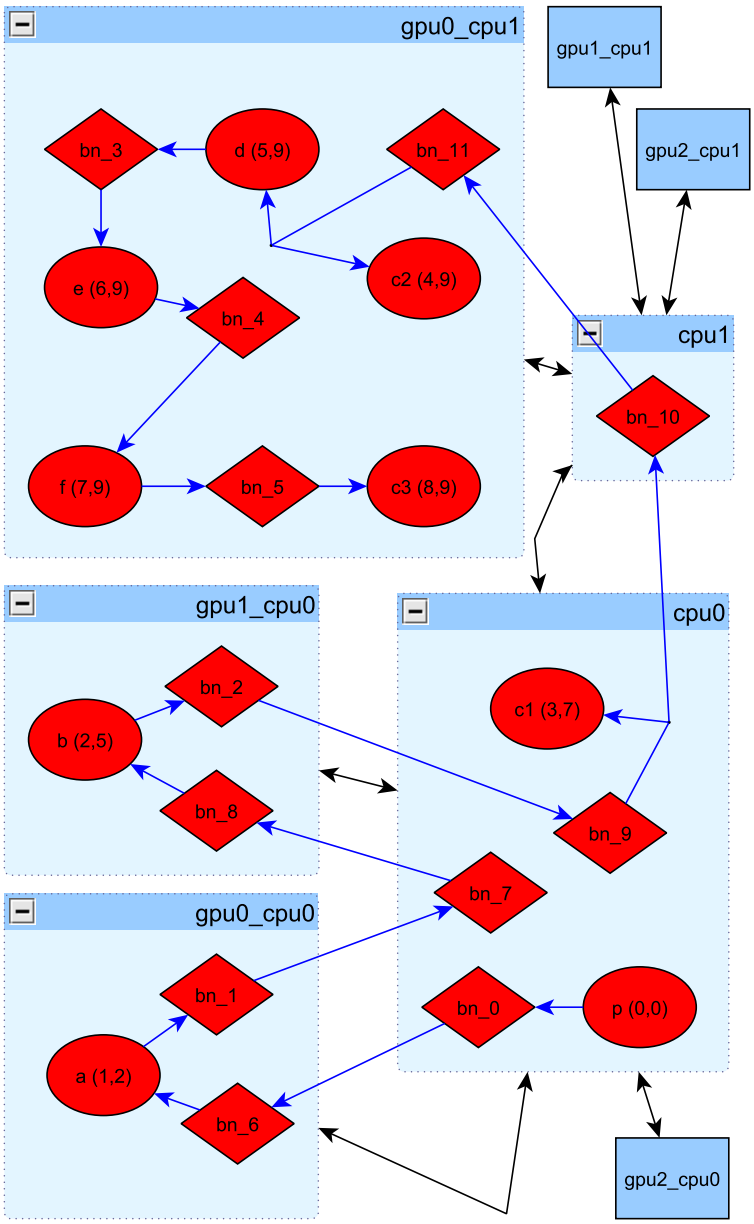


FIGURE 6.5 – Insertion de buffers pour le cheminement des données (jetons) entre les fonctions (acteurs) sur le cluster. Pour chaque acteur, une étiquette comprenant son nom et un couple de valeurs est associée. La première valeur correspond au rang de l'acteur fourni à l'ordonnancement. La seconde valeur correspond à la latence (ici en mode asynchrone).

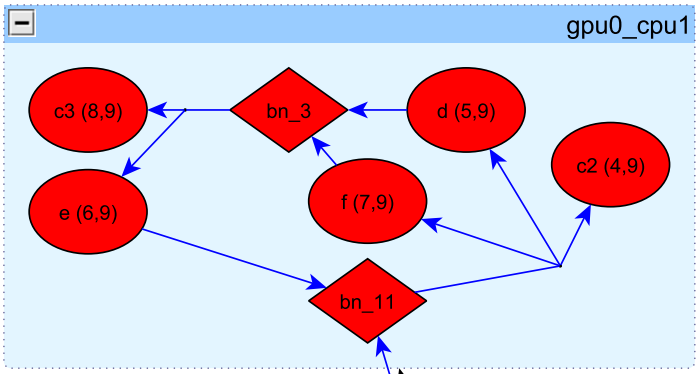


FIGURE 6.6 – Optimisation du nombre de buffers alloués grâce à un algorithme de coloriage de graphe.

Taille du buffer

Cette étape consiste à déterminer la taille des buffers : simple ou double. Le choix de la taille d'un buffer dépend de la stratégie de communication (avec ou sans recouvrement) et de son emplacement par rapport aux éléments de calcul environnant : si le mode d'exécution choisi est asynchrone et qu'un buffer reçoit/envoie des jetons à un élément de calcul différent du sien, la taille d'allocation doit être double. La stratégie de communication influe également sur le mode d'exécution (cf. section 6.1.3).

Allocation optimale des buffers mémoire

Une passe d'optimisation permet de partager des buffers mémoire au sein du même élément de calcul. Pour cela, un algorithme de coloriage de graphe colore les buffers qui pourraient être fusionnés de la même couleur ; le but étant de réduire le nombre de couleurs dans le graphe. L'algorithme de coloration possède une seule entrée : un graphe d'incompatibilité des buffers. Ce graphe d'incompatibilité, contenant des informations sur la capacité des buffers à être fusionnés, est produit à partie d'une règle simple : deux buffers sont incompatibles (ne peuvent pas être fusionnés) s'ils sont connectés au même acteur. Autrement, le même buffer serait simultanément utilisé en lecture et écriture (risque de pertes de données).

En outre, lorsque deux ou plusieurs buffers sont fusionnés, c'est le buffer de plus grande taille mémoire qui est alloué. La Figure 6.6 montre le résultat de cette optimisation appliquée à l'exemple d'application précédemment présenté. Dans le cas d'un recouvrement calculs-communications, le double buffer `bn_1` est lu d'un côté par `c2,d,f` tandis que `e` écrit de l'autre côté. De même, le double buffer `bn_3` est lu d'un côté par `e,c3` tandis que `d,f` écrit de l'autre côté. Cette optimisation a permis de libérer la mémoire allouée par deux buffers.

Calcul de la latence

Que le mode de recouvrement calculs-communications soit adopté ou pas, des cycles de retard sont introduit entre les éléments de calcul à cause de la synchronisation globale. Tandis que sur le même élément de calcul, les acteurs sont exécutés séquentiellement en une seule itération, d'un élément de calcul à l'autre, le flux des données subit une interruption lors de la synchronisation du système global. Ainsi, à un instant donné, les acteurs peuvent travailler sur différentes itérations de la même application simultanément. Par ailleurs, pour éviter que les acteurs qui n'ont pas encore reçu leur jeton soient déclenchés et que des données non valides ne circulent, nous calculons la latence en cycles (itérations) nécessaires avant que le jeton ne soit fourni en entrée. Les *threads* n'exécutent la fonction implémentée derrière l'acteur que quand le nombre de cycles est égale à la valeur de la latence.

- en synchrone (sans recouvrement calculs-communications) : chaque paire de buffers connectés introduit un cycle de retard.
- en asynchrone (recouvrement calculs-communications) : chaque double bubber introduit un cycle de retard supplémentaire.

6.1.3 A l'exécution

A l'exécution, sur chaque noeud de calcul du cluster, un *thread POSIX* est associé à chaque élément de calcul. Chaque *thread* gère les transferts et les lancement de fonction séquentiellement ou simultanément suivant le mode d'exécution choisi. Il suffit à chaque *thread* d'interroger le Graphe d'Implementation pour connaître les acteurs qui lui sont attribués : les acteurs mappés sur son élément de calcul lui sont attribués.

Il existe deux modes d'exécution. Chacun s'exécute itérativement suivant un des deux cycles suivants :

1. sans recouvrement calculs-communications
   - lancement des transferts CPU-CPU,
   - attente de la fin des transferts CPU sur tout le cluster,
   - lancement des transferts CPU-GPU,
   - attente de la fin des transferts GPU sur tout le cluster,
   - sur chaque élément de calcul, exécution séquentielle des acteurs en fonction de leur ordre d'ordonnancement ; exécution simultanée des éléments de calcul,
   - attente de la fin des calculs sur tous les éléments de calcul.
2. avec recouvrement calculs-communications
   - lancement asynchrone des transferts de/vers tous les éléments de calculs ET déclenchement des acteurs sur les éléments de calcul,
   - attente de la fin de toutes les tâches précédentes.

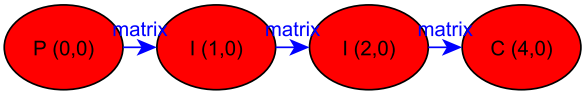


FIGURE 6.7 – DFG de l'application utilisée pour l'expérience sur cluster.

6.2 Cas d'étude

Dans cette section, nous présentons trois expériences réalisées avec notre outil. La première démontre les capacités de déploiement d'une application sur un cluster multi-GPU avec recouvrement calculs-communications. La seconde traite du portage multi-GPU de l'application de ganulométrie 3D. Enfin, la troisième partie présente le gain en performance suite à un déploiement scalable d'une application *streaming* de calcul des cartes de saillance visuelle.

6.2.1 Expérience de déploiement sur plusieurs noeuds de calcul

Cette section présente une expérience démontrant l'efficacité de notre flot de conception pour l'implémnetation aiséé d'une application test sur cluster multi-GPU. Le cluster en question est composé de deux hôtes connectés par un adaptateur Infiniband Quad Data Rate (QDR) x4 sur des ports PCI-E v2 x8. Le débit théorique maximal d'une ligne QDR est de 10 Gb/s (tous les 10 bits envoyés contiennent 8 bits de données). Ainsi, avec un adaptateur QDR x4 (quatre lignes), le débit maximal est de 40 Gb/s (soit 4 Go/s efficace). Nous avons mesuré un taux de transmission des données utiles de 2,6 Go/s, ce qui est conforme aux données du fabricant [1].

Chaque hôte possède une carte GPU GTX 285 connectée à un slot PCI-E v2 x16. Le bus PCI-E v2 offre un débit maximal de 500 Mo/s par ligne, ce qui conduit à une bande passante maximale de 8 Go/s en 16 lignes (x16). En pratique, nous avons mesuré des taux de transmission de données utiles de 5 Go/s.

L'application qui a été utilisé pour cette expérience est un DFG composé de quatre acteurs s'échangeant des matrices, cf. Figure 6.7. L'acteur-producteur P génère des matrices qu'il envoie à un premier acteur-incrémenteur I qui incrémente chaque élément de la matrice reçue. Cet acteur-incrémenteur renvoie ensuite son résultat à un second acteur-incrémenteur I. La matrice produite par ce dernier acteur est finalement envoyé à l'acteur-consommateur C où les résultat sont vérifiés de manière fonctionnelle.

L'application a été mappée sur le cluster comme indiqué sur la Figure 6.8.

1. Cf. Page 59 du "QLogic Fabric Software Installation Guide 7.0, version 7.0, Rev B"

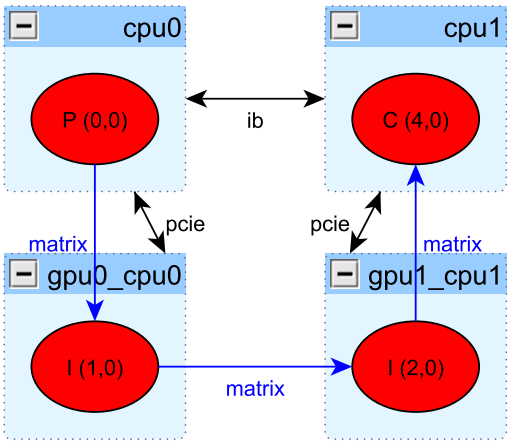


FIGURE 6.8 – Mapping du DFG sur le cluster.

Rappelons que notre objectif est de démontrer la capacité de notre outil à implémenter une application avec/sans recouvrement calculs-communications. Pour cela, les conditions expérimentales ont été définies de telle sorte que les temps de calcul soient du même ordre de grandeur que ceux de la communication. Afin que les étapes de création (producteur) et de vérification (consommateur) n'interfèrent pas avec les temps de calcul (incrémenteurs) et de communication, nous avons choisi de ne pas les introduire parmi les acteurs de production et de consommation. La charge de calcul des acteurs-incrémenteurs I peut être modifiée par l'utilisateur en jouant avec l'arguement `nb_loop` du kernel GPU (cf. Listing 6.3) qui implémente la série convergente suivante (elle converge vers $0,5$ et permet de calculer la valeur 1 pour l'incrémentation) :

$$\sum_{i=2}^{i \leq Nb\_loop+3} \frac{1}{i^2}$$

Listing 6.3 – Ce kernel prend en entrée la matrice *in* et produit en sortie la matrice *out*. Les paramètres *width* et *height* spécifient la taille de la matrice. L'argument *nb* $_{l}$*oop* permet de faire varier la charge de calcul en faisant varier la limite supérieure de la série convergente. Ce kernel n'a aucun autre but que de permettre à l'utilisateur de modifier le temps de calcul sur GPU

```
__global__ void kernel_matrix_inc(
    float* in,
    float* out,
    int width,
    int height,
    int nb_loop) {

    unsigned int x =
        blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y =
```

```
        blockIdx.y*blockDim.y + threadIdx.y;

    float one = 0.0f;
    int i;
    for( i = 2; i < nb_loop+3; i++ ) {
        one += (1.0f/i)*(1.0f/i);
    }
    one = (float)( (int)(one + 0.5f) );

    int index = y*width+x;
    out[index] = in[index]+one;
}
```

Nous avons comparé les performances de cette application avec et sans recouvrement calculs-communications pour une taille de matrice équivalent à 16 Mo ; les valeurs prisent par `nb_loop` varient dans la gamme $[0, 400]$. La Figure 6.9 montre l'allocation des buffers opérée par l'outil dans les deux cas avec et sans recouvrement calculs-communications. S'il n'y a pas de recouvrement, l'outil instancie $2 \times 16 = 32$ Mo de mémoire dans la mémoire RAM de chaque hôte (mémoire principale) ainsi que sur le GPU. Sinon, la taille des buffers alloués est doublée.

La courbe "without overlapping" (sans recouvrement calculs-communications) de la Figure 6.10 indique, sur l'échelle des ordonnées de gauche, le temps nécessaire (en millisecondes) pour terminer une itération de l'algorithme en fonction de la valeur de `nb_loop`. La courbe "with overlapping" montre les mêmes résultats avec recouvrement calculs-communications. La courbe "speedup" montre, sur l'échelle des ordonnées de droite, l'accélération calculée comme étant le rapport entre le temps sans et le temps avec recouvrement.

La courbe "without overlapping" est une ligne droite. Cela paraît logique puisque les calculs et les communications sont séquentialisés (et donc additionnés) et la complexité de l'algorithme exécuté par le GPU est en $O(nb\_loop)$. De façon plus formelle, si on note :

- T_{ib} le temps de transfert d'une matrice sur InfiniBand,
- T_h les temps de transferts CPU→GPU et CPU$leftarrow$GPU d'une matrice (effectués en parallèle),
- α la pente de la droite,
- T le temps d'une itération

En supposant que les temps d'initialisation de l'outil pour le lancement des transferts et des kernels sont négligeables par rapport aux temps de transfert, nous pouvons écrire :

$$T = T_h + T_{ib} + \alpha * nb\_loop$$

Comme nous pouvions nous y attendre, la courbe "with overlapping" commence par une

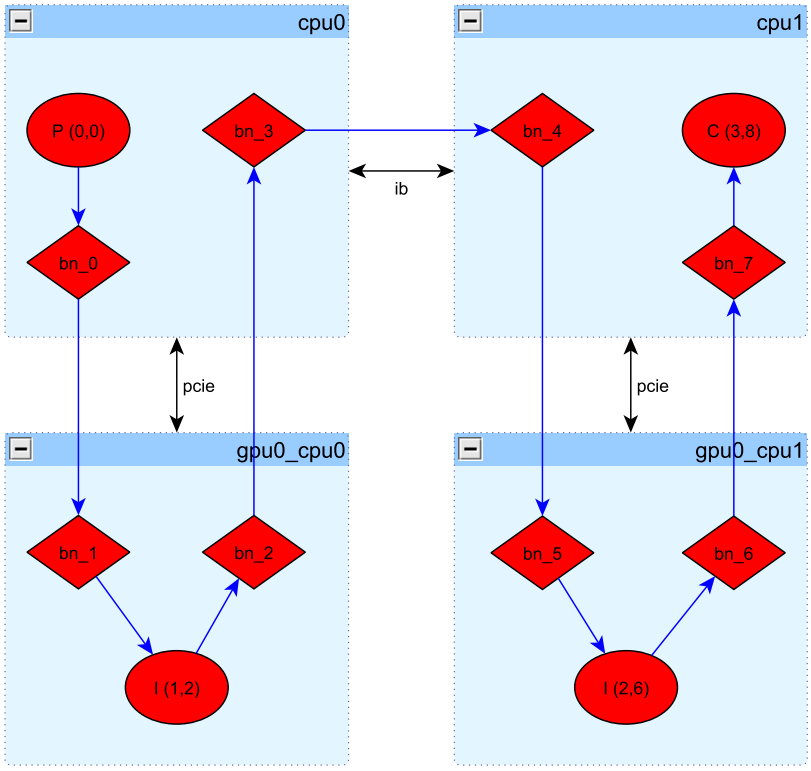


FIGURE 6.9 – Buffers mémoire alloués pour l'expérience sur cluster.

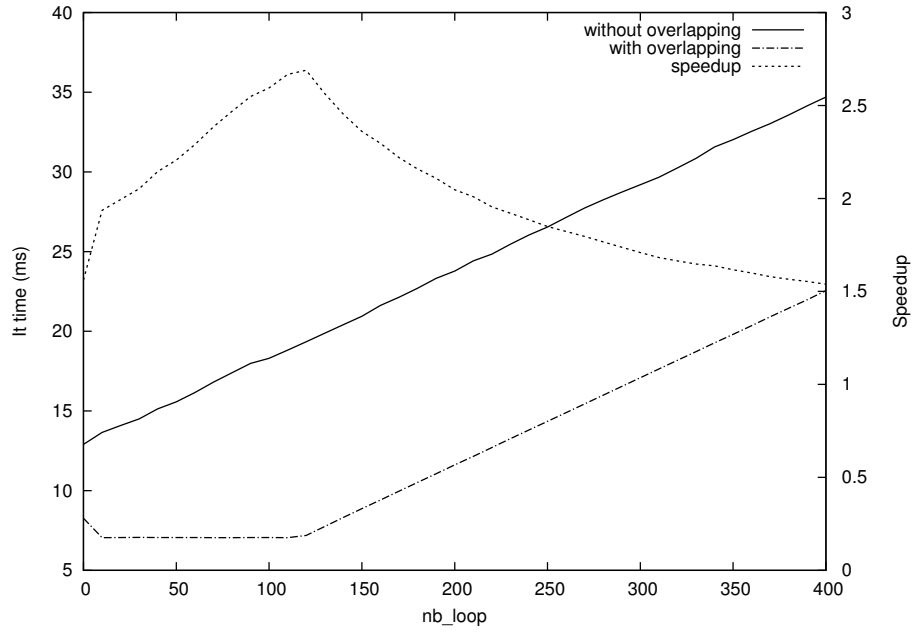


FIGURE 6.10 – Les résultats de l'expérience sur cluster montrent la configuration pour une accélération optimale avec recouvrement calculs-communications : les temps de calcul doivent être égaux aux temps de communication.

ligne horizontale, qui correspond aux valeurs de `Nb_loop` pour lesquelles le temps de calcul est plus court que les temps de communication. Puis, lorsque le temps de calcul est plus grand que les temps de communication, la courbe "with overlapping" devient une droite parallèle à la courbe "without overlapping". Puisque les transferts CPU-GPU et les communications CPU-CPU sont parallélisés, l'expression de la courbe "with overlapping" est :

1. Si $\alpha * nb\_loop < \max(T_{ib}, T_h)$
$$T = \max(T_{ib}, T_h)$$
2. Si $\alpha * nb\_loop \geq \max(T_{ib}, T_h)$
$$T = \alpha * nb\_loop$$

L'expression de l'accélération S, qui correspond à la courbe "speedup", est donc :

1. Si $\alpha * nb\_loop < \max(T_{ib}, T_h)$
$$S = \frac{T_{ib} + T_h + \alpha * nb\_loop}{\max(T_{ib}, T_h)}$$
Il s'agit d'une équation linéaire dont la valeur maximale est atteinte lorsque $\alpha * nb\_loop = \max(T_{ib}, T_h)$ est égal à :
$$\frac{T_{ib} + T_h}{\max(T_{ib}, T_h)} + 1$$

Nous pouvons voir que l'accélération est donc au plus égale à 3 si $T_{ib} = t_H$, ce qui se produit lorsque le temps de calcul est égale à la durée de communication sur InfiniBand qui est aussi égal à la durée de communication entre CPU et GPU.

Dans notre cas, si l'on note :
- R_{ib} le débit des données sur l'InfiniBand,
- R_h le débit des données entre CPU et GPU,
- d la taille de la matrice

l'expression de S devient (sachant que le débit des données le plus bas sur l'InfiniBand) :

$$S = \frac{\frac{d}{R_{ib}} + \frac{d}{R_h}}{\frac{d}{R_{ib}}} + 1 = 2 + \frac{R_{ib}}{R_h}$$

Ce qui équivaut à $2 + \frac{2.6}{5} = 2.5$ avec les figures fournies au début de cette sous-section. Cette accélération est en ligne avec la courbe d'accélération de la Figure 6.10 qui présente un maximum égal à 2,7 à $nb\_loop = 120$.

2. Si $\alpha * nb\_loop > \max(T_{ib}, T_h)$

$$S = \frac{T_{ib} + T_h}{\alpha * nb\_loop} + 1$$

dont l'asymptote à $+\infty$ égale à 1.

Pour conclure cette expérience, nous avons obtenu des résultats conformes à ceux attendus. En outre, notre outil permet de passer d'un mode avec recouvrement calculs-communications à un mode sans en modifiant un simple paramètre dans l'appel au programme. De même, il est tout aussi simple de modifier le mapping des acteurs du DFG. Grâce à l'outil, l'exploration architecturale est simplifiée et encouragée. Du point de vue des performances, nous avons observé que le recouvrement calculs-communications, nécessitant deux fois plus de mémoire, est le plus intéressant lorsque les temps de communication sont du même ordre de grandeur que les temps de calcul.

6.2.2 Granulométrie

Dans la section précédente, le portage d'une application test sur un cluster multi-GPU a permis de valider notre flot de conception. Cette section aborde l'implémentation multi-GPU de l'application de granulométrie 3D.

Le Graphe d'Implémentation

A partir de la décomposition présentée dans la Figure 4.4b, les éléments de calcul sont facilement identifiables : (1) le *thread* CPU lit l'image à traiter sur le disque dur et (2) le *thread*

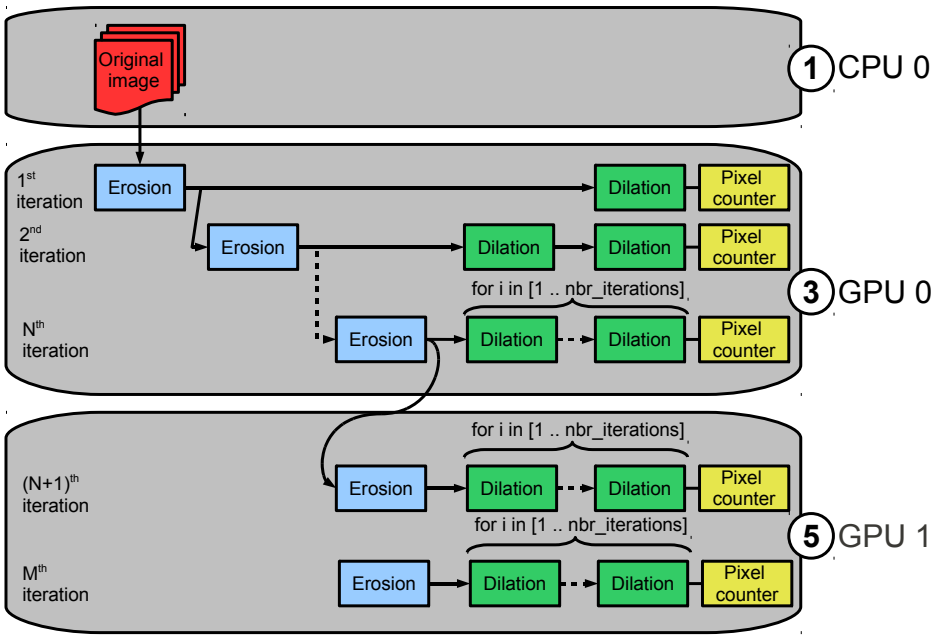


FIGURE 6.11 – L'application de granulométrie est répartie sur trois éléments de calcul.

GPU calcule les ouvertures jusqu'à ce que l'image soit vide. Cependant, pour une certain taille d'image et un certain nombre d'ouvertures le temps de calcul GPU devient prépondérant par rapport aux temps de lecture sur CPU et aux temps de communication.

Bien que cette application ne soit pas adaptée à une utilisation en pipeline, elle permettra d'illustrer les capacités d'équilibrage de charge de notre outil. Pour cela, nous déploierons l'application sur deux GPU ; ce sera également l'occasion de démontrer que le parallélisme de données (cf. Chapitre 4) est mieux adapté à l'application de granulométrie qu'un parallélisme en pipeline.

L'application répartie sur trois éléments de calcul (Figure 6.11) nécessite l'accomplissement de six tâches (Figure 6.12) :

1. le *thread* CPU lit l'image à traiter sur le disque dur,
2. le transfert CPU→GPU 0 de l'image,
3. le calcul des N premières ouvertures sur GPU 0 (N est fixé par l'utilisateur),
4. le transfert du résultat intermédiaire (image + données résultat) de GPU 0 à GPU 1 (GPU 0→CPU puis CPU→GPU 1),
5. le calcul des M dernières ouvertures sur GPU 1,
6. le transfert du résultat final de la granulométrie sur le CPU.

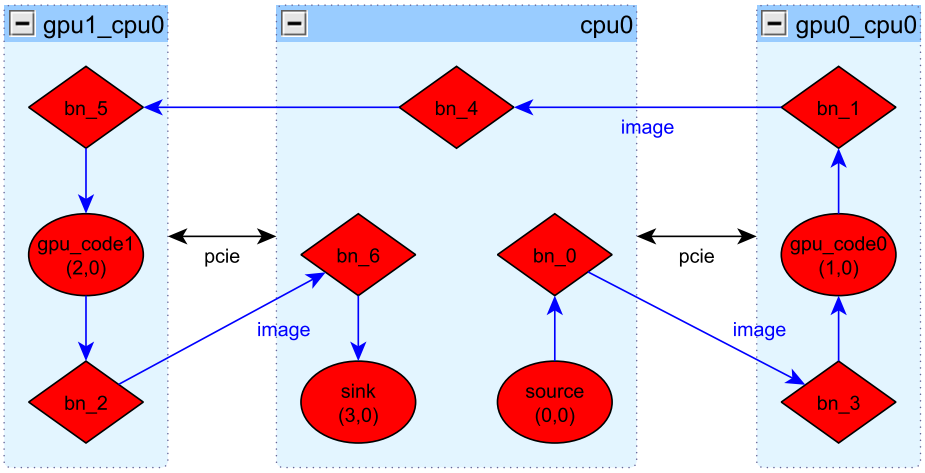


FIGURE 6.12 – Graphe d'Implémentation de l'application de granulométrie.

Résultats

La charge de travail entre les deux GPU peut être réglée par l'utilisateur en modifiant simplement les valeurs N et M. Ces deux paramèters sont modifiables directement un fichier de paramètres parsé à l'exécution et contenant des informations comme : le nombre d'ouvertures, la taille de l'image, l'emplacement de l'image, l'emplacement du fichier résultat, la taille de bloc de *threads*, etc

La Figure 6.13 présente les résultats pour différentes répartitions de la charge de travail sur les GPU avec recouvrement calculs-communications. Dans cet exemple, l'image requiert 40 ouvertures pour filtrer toutes les tailles d'objet. La répartition de ces ouvertures sur les deux GPU est optimale lorsque les ouvertures 1 à 28 sont effectuées sur le premier GPU (GPU 0) alors que les ouvertures 29 à 40 sont effectuées sur le second GPU (GPU 1) : le gain obtenu par rapport à une exécution séquentielle s'élève à x2.7 grâce au recouvrement entre les calculs CPU et GPU et les communications.

6.2.3 Modèle pour le calcul des cartes de saillance visuelle

Cette section décrit l'implémentation d'une application de calcul intensif inspirée d'un modèle biologique. Basé sur la rétine du primate, le modèle de saillance visuelle est utilisé pour localiser les régions d'intérêt *i.e.* la capacité qu'a la vision humaine de pouvoir concentrer son attention sur des régions spécifiques d'une scène visuelle. Nous proposons d'implémenter ce modèle sur un noeud multi-GPU avec recouvrement calculs-communications.

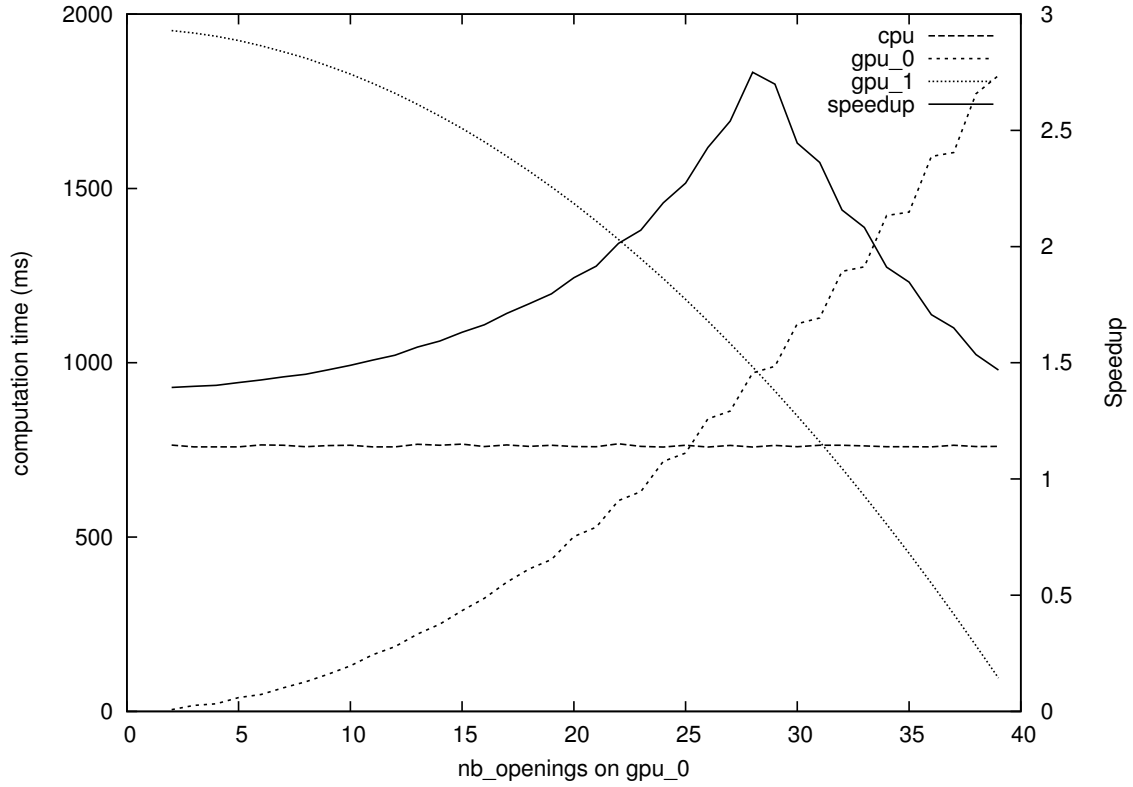


FIGURE 6.13 – Equilibrage de charge sur deux GPU avec recouvrement calculs-communications. Le temps de lecture (CPU) du volume 512^3 depuis le disque dur est constant.

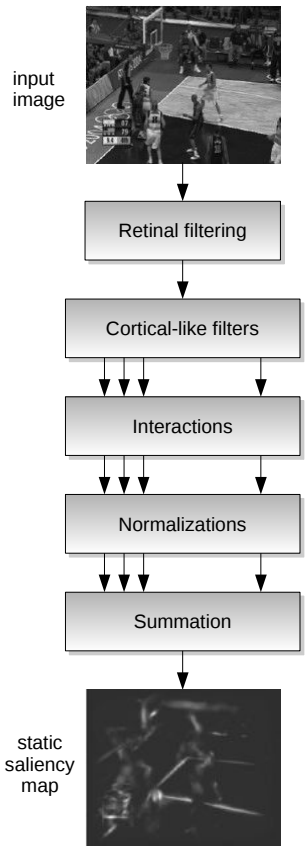


FIGURE 6.14 – Etapes de la voie statique du modèle de saillance visuelle.

Le modèle

Le modèle de saillance mis en oeuvre est celui proposé par [56] comme indiqué dans la Figure 6.14. On utilise des filtres de Gabor pour simuler la décomposition de l'information visuelle en différentes fréquences spatiales et orientations dans le cortex visuel primaire. Chaque filtre G_{ij}, d'orientation i et de fréquence j, est défini par sa fréquence centrale radiale f_j et son écart-type σ_{ij} (Figure 6.15).

$$G_{i,j}(u,v) = exp(-\frac{(u'-f_j)^2 + v'^2}{2{\sigma_{ij}}^2})$$

$$\begin{cases} u' = u \cdot cos(\theta_i) + v \cdot sin(\theta_i) \\ v' = v \cdot cos(\theta_i) - u \cdot sin(\theta_i) \end{cases}$$

La réponses des neurones dans le cortex primaire est influencée par d'autres neurones. Ces

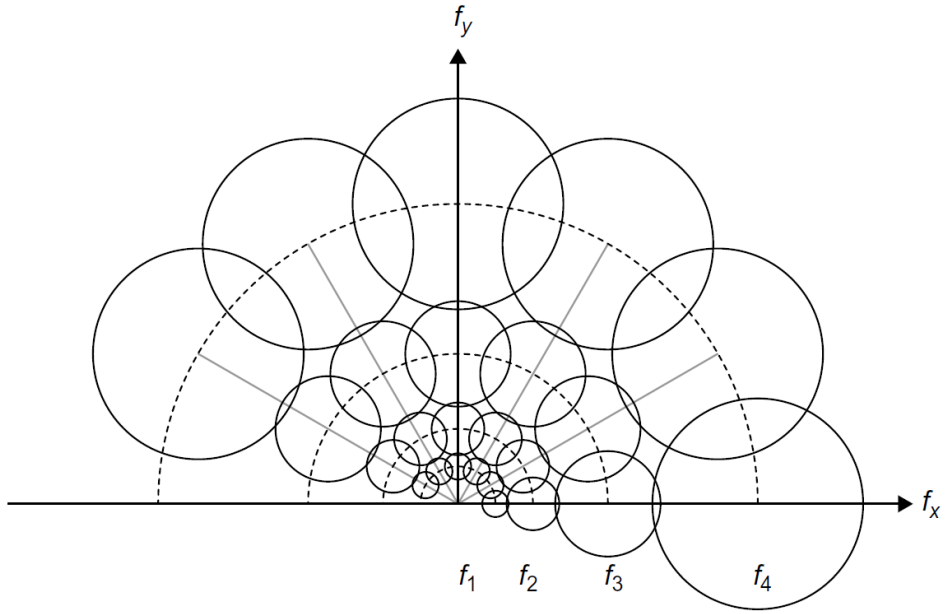


FIGURE 6.15 – Représentation par des filtres de Gabor du cortex visuel : jeu de 24 filtres G_{ij} avec 4 fréquences et 6 orientations.

interactions se produisent au niveau d'un pixel à travers différentes cartes partielles (partial maps) : les objets se trouvant dans la même orientation avec des fréquences différentes sont renforcés tandis que ceux situés à la même fréquence avec des orientations différentes sont diminués. Une région est saillante si elle est différente de ses voisins. Puis, après une première normalisation, chaque carte partielle est multipliée par un facteur $(max(m_{ij}) - \overline{m_{ij}})^2$: le maximum de la carte diminué de sa moyenne quadratique. Finalement, toutes les cartes partielles sont additionnées pour obtenir une carte de saillance statique. Cette approche statique est détaillée dans [71].

Implémentation de l'algorithme

L'Algorithme 2 détaille les étapes de l'implémentation. Tout d'abord, l'image d'entrée r_im est filtrée par un filtre de Hanning pour réduire l'intensité lumineuse sur les bords. Dans le domaine fréquentiel, cf_fim est traitée par un banc de filtre de Gabor 2D constitué de six orientations et quatre bandes de fréquences. Les 24 cartes partielles $cf\_maps[i, j]$ sont déplacées dans le domaine spatial $c\_maps[i, j]$. Les interactions courtes inhibent ou excitent les pixels en fonction de l'orientation et de la bande de fréquence des cartes partielles. Les valeurs obtenues sont normalisées selon une fourchette dynamique avant l'application de la méthode d'Itti pour la normalisation et la suppression des valeurs inférieures à un certain seuil. Enfin, toutes les cartes partielles sont accumulées dans une seule carte, qui est la carte de saillance visuelle de la voie statique [96, 95].

Algorithme 2 Voie statique du modèle de saillance visuelle

Entrées: Une image r_im de taille $w \cdot l$
Sorties: La carte de saillance visuelle

1: $r\_fim \leftarrow FiltreDeHanning(r\_im)$
2: $cf\_fim \leftarrow FFT(r\_fim)$
3: **pour** $i \leftarrow 1$ **to** orientations **faire**
4: **pour** $j \leftarrow 1$ **to** frequences **faire**
5: $cf\_maps[i,j] \leftarrow FiltreDeGabor(cf\_fim, i, j)$
6: $c\_maps[i,j] \leftarrow IFFT(cf\_maps[i,j])$
7: $r\_maps[i,j] \leftarrow Interactions(c\_maps[i,j])$
8: $r\_norm\_maps[i,j] \leftarrow Normalisations(r\_maps[i,j])$
9: **fin pour**
10: **fin pour**
11: $CarteDeSaillance = Sommation(r\_norm\_maps[i,j])$

Résultats

Le portage de cette application sur un seul GPU donne des résultats acceptables pour l'exploitation en temps réel. Pour une vidéo d'entrée de taille 512x512 pixels, le débit sortant est de 44 fps (frames per second). Cependant, pour une vidéo d'entrée de taille 720x576 pixels, le débit sortant chute à 21 fps. Afin d'obtenir un débit sortant respectant des contraintes de temps réel (30 fps) pour des vidéos en entrée de dimensions plus grandes que 512x512 pixels, l'application a été portée en pipeline sur deux GPU. Une exploration exhaustive de toutes les répartitions de tâche possibles sur les deux GPU nous a orienté vers le choix de la configuration optimale qui est la suivante : les cinq premières étapes (filtre de Hanning, FFT, filtre de Gabor, IFFT et Interactions) sont portées sur le premier GPU tandis que la dernière étape (Normalisations) est portée sur le second GPU.

La Figure 6.16 présente l'allure de la trace pour l'exécution de l'application de saillance visuelle avec la configuration optimale décrite ci-dessus sur une vidéo en entrée de dimensions 720x576 pixels. Lors d'une utilisation de GPU en pipeline, puisque ces derniers sont synchronisés entre chaque stade du pipeline, le débit sortant se cale sur le GPU le plus lent *i.e.* le débit sortant correspond au débit du GPU le moins performant. Dans le cas de la configuration précédemment choisie, le GPU 0 présente le débit sortant le plus faible, le débit sortant est donc équivalent au sien, soit 33 fps. A noter que bien que le temps d'occupation du GPU 0 soit de 100%, les traitements délégués au GPU 1 ne représentent que 55% de son temps d'occupation. Cette perte d'efficacité au niveau de l'utilisation des ressources de calcul est dûe à une granularité des tâches qui composent l'application pas assez fine pour permettre un équilibrage de la charge de travail par GPU qui soit optimal. Celle-ci aurait pu être mieux équilibrée sur les deux GPU, si l'étape Interactions avait pu être divisée en plusieurs sous-tâches de grain plus fin. En outre, il est intéressant de noter que les temps de transfert CPU $\leftrightarrow$GPU sont plus élevés dans cette configuration de répartition des tâches qu'avec un

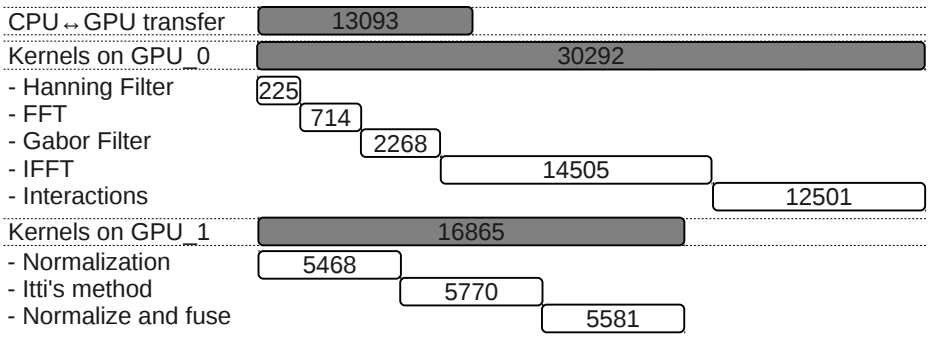


FIGURE 6.16 – Allure de la trace (en millisecondes) d'exécution d'une implémentation en pipeline sur deux GPU de l'application de saillance visuelle sur une vidéo de dimensions 720x576 pixels. Les transferts et les kernels de GPU 0 et GPU 1 sont exécutés simultanéments. En d'autres termes, les temps en gris foncés se recouvrent.

seul GPU. Cela est dû au transfert des 24 cartes partielles entre GPU 0 et GPU 1. Cependant, grâce au recouvrement calculs-communications, les performances ne sont pas affectées puisque ces transferts sont recouverts par les temps de calcul sur GPU. Notez que la quantité de buffers alloués sur GPU 0 (cf. Figure 6.17) est divisée par deux après la passe d'optimisation de la quantités de buffers alloués (cf. section 6.1.2).

6.3 Conclusion

Dans ce chapitre, nous avons présenté l'approche développée dans notre bibliothèque PACCOlib. Elle est basée sur une simplification de l'implémentation d'applications de TdSI sur cluster multi-GPU tout en permettant l'expression du parallélisme de tâches et le recouvrement calculs-communications. Grâce à notre flot de conception, le programmeur n'a pas besoin de gérer les communication inter-composants, la mémoire, le *multi-threading* et les synchronisations. Il ne perd pas de temps sur des opérations de codage basiques et rudimentaires mais souvent complexe à gérer colletivement : MPI, Pthreads, etc. Il se concentre plutôt sur le développement des codes de kernel de calcul. Aussi, une application développée pour une certaine configuration de cluster peut facilement être transposée sur une autre plateforme ; la répartition des tâches sur les éléments de calcul devient une opération simple à la portée du programmeur lambda. De plus, avec cet effort de déploiement de l'application sur cluster en moins, l'utilisateur peut se permettre d'explorer aisément les différentes configurations de répartition des tâches pour un équilibrage de charge du système optimal.

Tandis que les fabricants de GPU déploient des efforts importants pour rendre leur matériel de plus en plus adapté au calcul scientifique, les communications inter-éléments de calcul seront sans doute le prochain goulot d'étranglement pour la programmation. Par conséquent, nous pensons que l'exploitation efficace des ressources matérielles des architecture parallèles

Etape	Temps (μs)	Temps (%total)
filtre de Hanning	136	0.60
FFT	358	1.57
filtre de Gabor	1446	6.33
IFFT	6645	29.09
Interactions	4127	18.07
Normalisations		
-normalisation	3269	14.31
-méthode d'Itti	3453	15.12
-normalisation et fusion	3405	14.91
Total	22839	100

(a) 512×512

Etape	Temps (μs)	Temps (%total)
filtre de Hanning	225	0.47
FFT	738	1.55
filtre de Gabor	2276	4.77
IFFT	14908	31.23
Interactions	12510	26.21
Normalizations		
-normalisation	5554	11.64
-méthode d'Itti	5839	12.23
-normalisation et fusion	5681	11.90
Total	47731	100

(b) 720×576

TABLE 6.1 – Temps de calcul des différentes étapes de l'application de saillance visuelle pour différentes dimensions de vidéo.

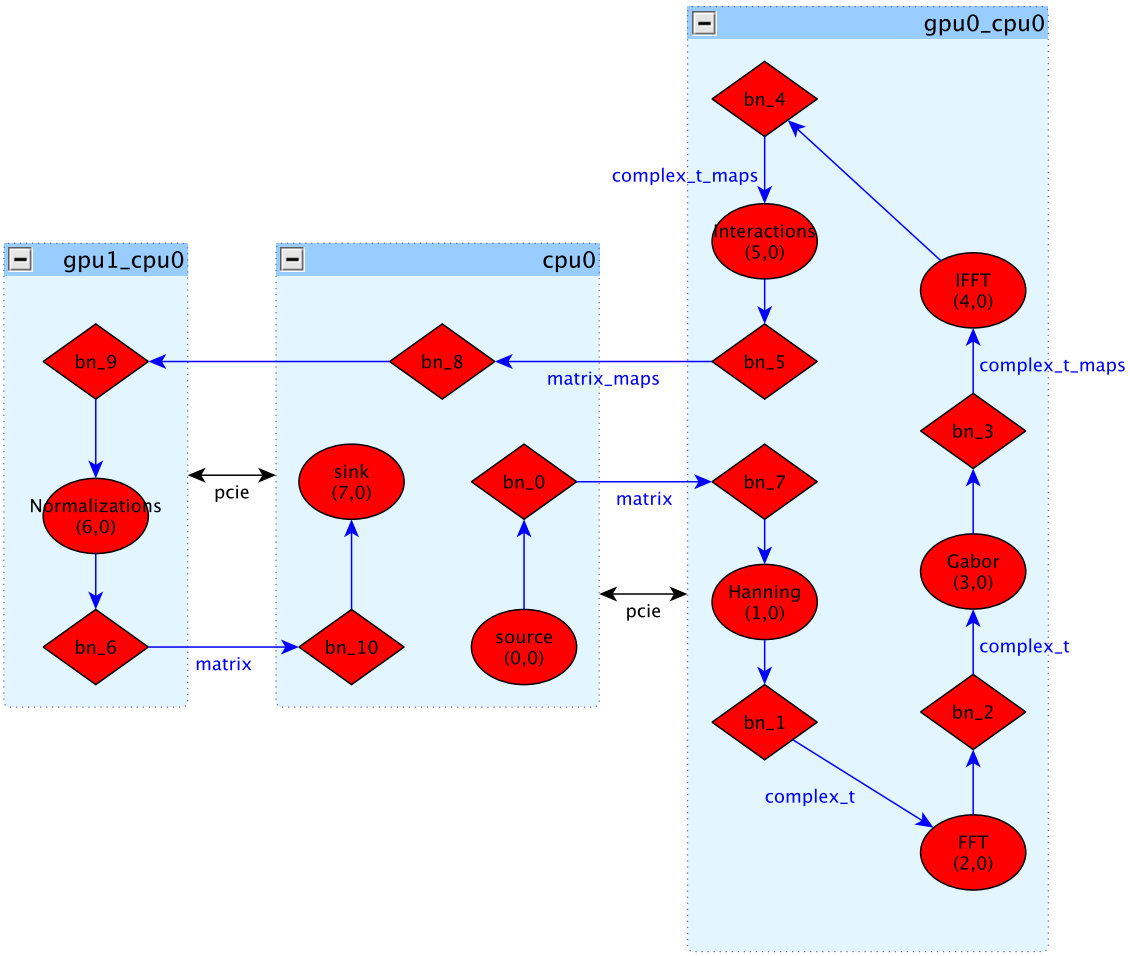


FIGURE 6.17 – Graphe d'Implémentation du modèle de saillance visuelle.

émergentes passera indubitablement par un intérêt grandissant pour la gestion efficace des communications et des données entre les différents espaces mémoire.

Grâce à notre outil, l'application de saillance visuelle qui nécessitait un gain en performance supplémentaire à celui obtenu sur mono-GPU a été atteint sur deux GPU. En effet, le portage de l'application a été effectué en pipeline sur deux GPU situés sur un noeud CPU identique. La mise a niveau a été réalisée aisément grâce à l'automatisation des allocations mémoires, du cheminement des données entre les deux GPU et du recouvrement calculs-communications. De plus, avec de simples annotations de mapping, l'équilibrage de charge entre les deux GPU a été effectué aisément améliorant significativement le débit sortant de 21 fps à 33 fps pour le traitement d'une vidéo de résolution 720x576 pixels.

Par ailleurs, certaines personnes pourraient se demander : pourquoi avons-nous choisi d'implémenter l'application de saillance visuelle en pipeline ?

6.3.1 Discussion autour du parallélisme temporel (ou bien parallélisme de pipeline)

"Au lieu de porter l'application en pipeline, n'aurait-il pas été plus simple d'envoyer une *frame* sur deux à chaque GPU ?"

Finalement, la question est de choisir entre du *task-farm parallelism* (SPMD) et du parallélisme fonctionnel (MPMD). Dans le premier cas, les données sont réparties sur les unités de calcul exécutant le même programme. Dans le second cas, les données sont envoyées en pipeline sur les unités de calcul exécutant des tâches différentes. Dans le premier cas, une tâche doit s'occuper de répartir les données sur les éléments de calcul puis une autre de les récolter et les réordonner en sortie. Dans le second cas, aucune tâche supplémentaire ne doit être développée pour la distribution et la récolte des résultats, seule la répartition des tâches sur les éléments de calcul est à la charge du programmeur.

Dans la suite de ce paragraphe, nous présenterons les arguments pour et contre ces deux types de parallélisme.

D'abord, abordons l'usage du parallélisme de pipeline dans notre outil :

- Pour :
  - le débit sortant est constant - à condition que les temps de calcul soient indépendants des valeurs des données traitées (ce qui est le cas de l'application de saillance visuelle présentée).
  - les résultats sortants sont délivrés dans le bon ordre.

  - aucune modification de code, développement de kernel supplémentaire n'est nécessaire.
- Contre :
  - Quand la charge de travail est mal répartie entre les GPU, l'utilisation des ressources matérielles devient non efficace. Il est possible d'y remédier si la décomposition des tâches de l'application est suffisamment fine pour permettre une répartition homogène - ce qui n'est pas toujours le cas.

Ensuite, exposons les arguments pour et contre une approche *task-farm parallelism* :

- Contre :
  - un mécanisme de décomposition et de distribution des données devra être mis en place. Ce mécanisme peut être plus ou moins complexe selon l'intensité et la répartition des liens de dépendance entre les données. Dans notre cas, le traitement de la vidéo par *frames* successives indépendamment des *frames* précédentes simplifie la décomposition et la distribution des données par l'implémentation d'une pile de *frames* dans laquelle chaque GPU vient piocher.
  - un mécanisme de récupération et de réordonnancement des données résultats devra être implémenté. A priori, il est difficile d'assurer que l'ordre dans lequel les données résultats provenant des GPU seront délivrées corresponde à l'ordre dans lequel les données d'entrée ont été distribuées - cela dépendra du débit sortant des GPU, des *overheads* du runtime, etc.
  - sans re-synchronisation des données résultats, le débit de la séquence vidéo sortante sera sans doute non régulier et peu rendre l'affichage non agréable à l'oeil nu.
- Pour :
  - le débit sortant pouvant varier d'un GPU à l'autre, cette approche permet d'éviter de gaspiller des ressources matérielles à cause d'une répartition non homogène de la charge de travail sur les GPU. Les GPU sont exploités de façon optimale puisque leur exécution est guidée par le flux de données.

Ainsi, l'avantage d'un portage en pipeline est que l'utilisateur n'a pas besoin de gérer la répartition des données, leur récupération et le ré-arrangement des données dans leur ordre d'origine. Aussi, il n'aura pas besoin de re-synchroniser le débit sortant pour le rendre régulier. En outre, certaines applications nécessitent une implémentation en pipeline. Typiquement, des applications avec des contraintes de temps et des dépendances de données inter-itérations *i.e.* des fonctions qui possède un état interne.

Chapitre 7

Migration de tâches

Sommaire

7.1 Motivations 135
7.2 Implémentation 136
7.3 Résultats 141
7.4 Conclusion 145

7.1 Motivations

La méthode de migration qui sera détaillée dans la suite de ce chapitre vient en appui à l'outil PACCOlib. En effet, le mapping fourni par le programmeur dans PACCOlib est statique. En d'autres termes, l'assignation des tâches sur les éléments de calcul se fait *hors ligne*. Il n'y a donc aucun moyen pour le programmeur de modifier l'emplacement d'une tâche une fois que l'application est lancée. La migration de tâche est une opération *en ligne* qui rend cette modification possible. Le but consiste à faire migrer une tâche de l'élément de calcul sur lequel elle s'exécute vers un autre élément de calcul cible.

La liste suivante présente les principaux intérêts d'un mécanisme de migration de tâche :
- Un équilibrage de charge dynamique permettant d'assurer une utilisation efficace des ressources en fonction du facteur d'optimisation choisi : la vitesse d'exécution du programme, l'efficacité énergétique, etc
- Un arrêt volontaire de la machine pour cause de maintenance.
- Une tolérance aux fautes au cas où un élément de calcul se met à dysfonctionner. A noter que nous ne traitons pas cet aspect dans notre approche car il nécessite un mécanisme de détection de faute que nous n'avons pas mis en place.

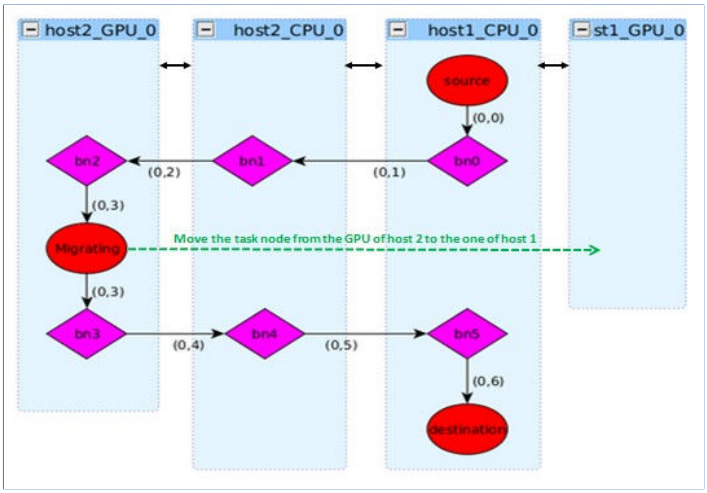


FIGURE 7.1 – Exemple de cas d'étude pour la migration de tâches. Un nouveau GPU *host1_GPU_0* a été monté sur la machine *host1_CPU_0*. La tâche initialement exécutée sur *host2_GPU_0* peut désormais être migrée sur *host1_GPU_0*. Le chemin des communications sera ainsi réduit.

En résumé, la migration de tâche permet à l'utilisateur de mieux gérer son parc informatique.

Prenons par exemple le cas d'un cluster constitué de deux CPU et d'un GPU. Une application GPU doit être lancée sur ce cluster. Cependant, le CPU hôte, *i.e.* celui sur lequel l'application est lancée, ne dipose pas de GPU. Comme illustré sur la Figure 7.1, les données à traiter doivent alors parcourir le cluster pour être exécutées sur le GPU de l'autre CPU. Les données suivent le parcours suivant : issues de *host1_CPU_0*, elles passent par *host2_CPU_0* avant d'être transmises à *host2_GPU_0* et le trajet inverse parcours les mêmes éléments de calcul. Maintenant, un nouveau GPU *host1_GPU_0* a été monté sur la machine *host1_CPU_0*. Le chemin emprunté par les données étant plus court entre *host1_CPU_0* et *host1_GPU_0* (un lien PCI-E) qu'entre *host1_CPU_0* et *host2_GPU_0* (un lien InfiniBand + un lien PCI-E), il est préférable de migrer la tâche initialement lancée sur *host2_GPU_0* sur le nouvel élément de calcul *host1_GPU_0*. Cela permettra de réduire les temps de communication et d'éviter une éventuelle congestion du réseau si d'autres applications sont lancées simultanément sur le cluster.

7.2 Implémentation

Afin d'incorporer la méthode de migration dans notre outil, il faut l'introduire parmi les étapes du processus d'exécution de PACCOlib. Pour rappel, l'exécution de l'outil est itérative et chaque itération est constituée, dans le cas d'un fonctionnement synchrone, de trois étapes (plus la synchronisation globale du système) : les transferts CPU-CPU, les transferts CPU-GPU

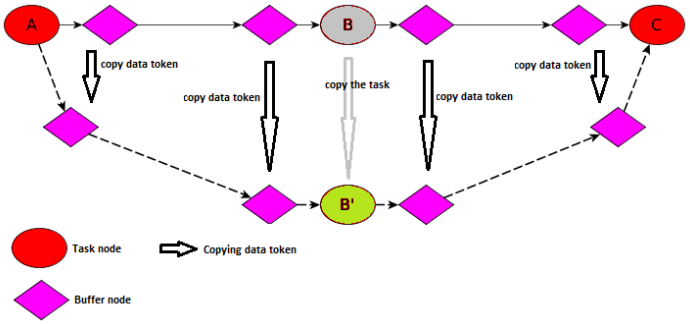


FIGURE 7.2 – Cas présent dans la littérature : Ebner et al. [34] présentent une solution introduisant la migration de tâche en une seule passe.

et l'exécution des tâches sur les éléments de calcul. Dans le cas d'un fonctionnement asynchrone (avec recouvrement calculs-communications), ces trois étapes s'exécutent simultanément et le temps d'exécution d'une itération correspond alors au temps d'exécution maximal de l'une de ces trois étapes. On assigne un thread par élément de calcul et un thread supplémentaire par noeud de calcul pour assurer la synchronisation globale. En prenant le cas de la Figure 7.1 pour exemple, 6 threads sont créés pour gérer le cluster : 4 threads sont assignés aux éléments de calcul (2 CPU + 2 GPU) et 2 threads sont assignés aux deux noeuds du cluster.

Nous donc choisi d'insérer la méthode de migration après l'étape de synchronisation globale du système. Cela implique qu'une tâche ne peut être migrée durant son exécution ; la migration de tâche n'a lieu qu'à la fin de celle-ci. En d'autres termes, la méthode de migration de tâche implémentée ne permet pas la préemtion. Quant aux modifications des chemins de flux de données, ceux-ci sont rapportés dans le Graphe d'Implémentation. C'est le thread de synchronisation qui s'occupe de le mettre à jour. Les autres threads intègrent ensuite les modifications apportées au Graphe d'Implémentation.

Ebner et al. [34] présentent une solution introduisant la migration en une seule passe. La Figure 7.2 illustre les étapes de ce processus : allouer les buffers sur le "nouveau" chemin emprunté par les données (chemin du bas), transférer les données de chaque "ancien" buffer (chemin du haut) dans le "nouveau" buffer correspondant (chemin du bas) et, finalement, migrer la tâche vers son nouvel emplacement (chemin du bas). Cependant, selon la granularité des données et le type des communications (rapides/lentes), l'accumulation de ces étapes bout à bout peut devenir relativement longue. Il est alors possible que l'étape de migration empiète considérablement sur le déroulement "normal" de l'application. En effet, si la migration occupe trop de temps lors d'une itération de l'application, les éléments de calcul non concernés par la migration seront alors en attente et les capacités de calcul du système seront sous exploitées.

Nous avons choisi d'implémenter une méthode mettant en oeuvre une migration graduelle étalée sur plusieur itérations. Ainsi, à chaque itération, une action appartenant à la migration est effectuée. L'inconvénient de cette approche est la coexistance sur le Graphe d'Implémentation des deux chemins de données (ancien et nouveau) durant plusieurs itérations. Ce n'est qu'une

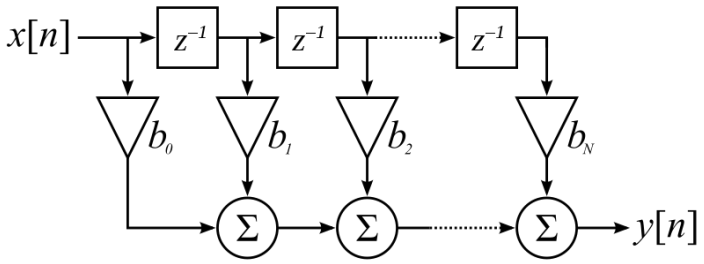


FIGURE 7.3 – Illustration d'un filtre FIR.

fois que la migration de tâche est terminée que l'"ancien" chemin, devenu obsolète, est supprimé du Graphe d'Implémentation.

Notre implémentation peut être décomposée en deux grandes étapes : l'allocation des buffers et la redirection du flux de données. La redirection du flux de données diffère de l'approche d'Ebner et al. par l'absence des transferts de données depuis les buffers de l'"ancien" chemin vers ceux du "nouveau" chemin. En effet, les données contenues sur l'"ancien" chemin continuent d'avancer dans le pipeline jusqu'à ce qu'il soit complètement vidé. En d'autres termes, une fois que l'opération de migration de tâche est engagée, le flux des données n'alimente plus le pipeline de données sur l'"ancien" chemin. Cependant, le contenu du pipeline continue d'avancer jusqu'à ce qu'il soit complètement purgé. Les buffers peuvent alors être désalloués et l'"ancien" chemin disparaît du Graphe d'Implémentation.

Cette seconde étape présente deux modes de fonctionnement qui dépendent du type de la tâche. Celle-ci peut être AVEC ou SANS état. Si l'exécution d'une tâche dépend d'informations récoltées durant les précédentes itérations de l'application, cette tâche possède un état interne. Cet état interne peut correspondre à la conservation dans un buffer interne des N données fournies aux itérations $n-1$ à $n-N$ ou à une variable utilisée dans un prédicat de condition de branchement ou d'arrêt de boucle. Prenons l'exemple d'un filtre FIR (cf. Figure 7.3). Son équation est :

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \tag{7.1}$$

$$= \sum_{i=0}^{N} b_i x[n-i] \tag{7.2}$$

Pour calculer sa valeur $y[n]$, il a besoin des valeurs lues pendant les N dernières itérations. Un filtre FIR est donc une tâche AVEC état.

La migration d'une tâche SANS état est plus simple puisque seules les données présentes dans son buffer d'entrée sont nécessaires à son exécution. Pour que la migration réussisse, il suffit que le flux des données ait été dévié vers l'entrée de la "nouvelle" tâche et que les données provenant des deux chemins "ancien" et "nouveau" arrivent dans le bon ordre en entrée de la tâche suivante.

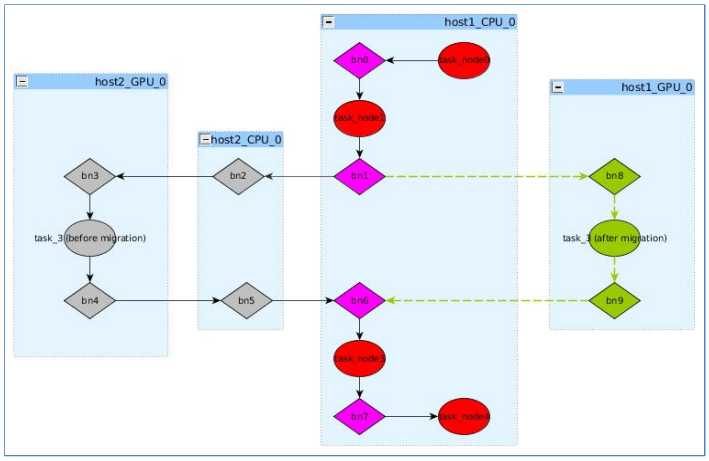


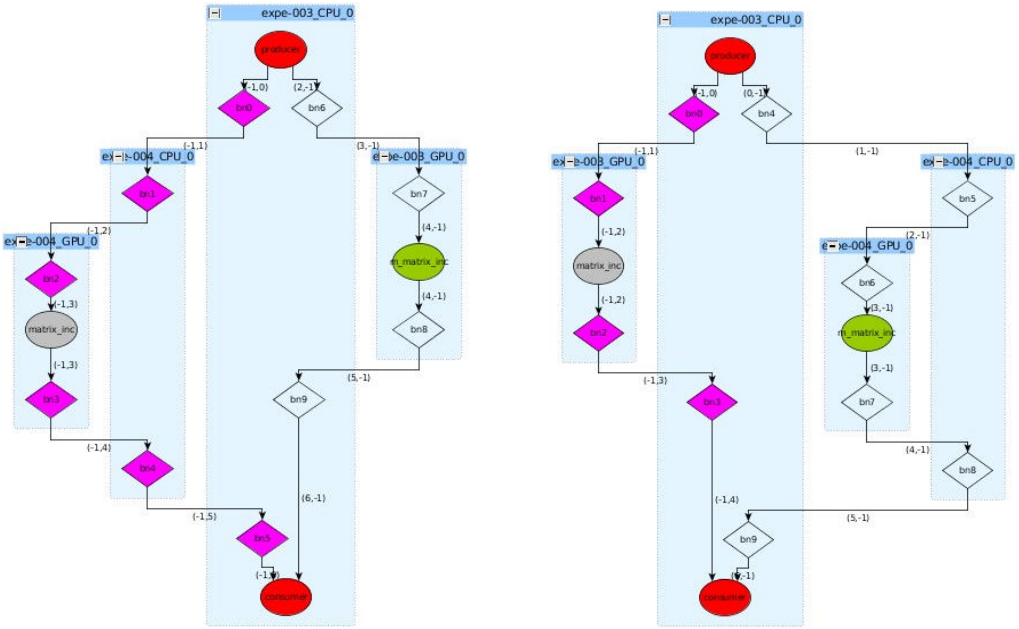
FIGURE 7.4 – Etape 1. Allocation de buffers supplémentaires. Le pipeline décédé est de couleur grise tandis que le pipeline ressuscité est de couleur verte.

Ci-dessous nous présentons la terminologie qui sera utilisée dans le reste de ce chapitre pour représenter les différents éléments entrant en jeu lors de la migration :

- tâche avec état : c'est une tâche qui présente certaines dépendances vis à vis de ses précédentes exécutions. Ce cas particulier apporte une contrainte supplémentaire à l'exécution de l'opération de migration.
- tâche sans état : c'est une tâche qui ne présente pas de dépendance vis à vis de ses précédentes exécutions. Quand le type de tâche (avec ou sans état) n'est pas précisé, il s'agit d'une tâche sans état.
- tâche décédée : c'est l'emplacement d'origine de la tâche qui doit être migrée.
- tâche réssuscitée : c'est l'emplacement d'origine de la tâche qui doit être migrée.
- pipeline décédé : c'est le chemin de buffers qui passe par la tâche décédée. Les buffers le constituant seront désalloués après que son contenu ait été purgé.
- pipeline ressuscité : c'est le chemin de buffers qui passe par la tâche ressuscitée. Les buffers le constituant sont alloués avant la redirection du flux de données par ce chemin.

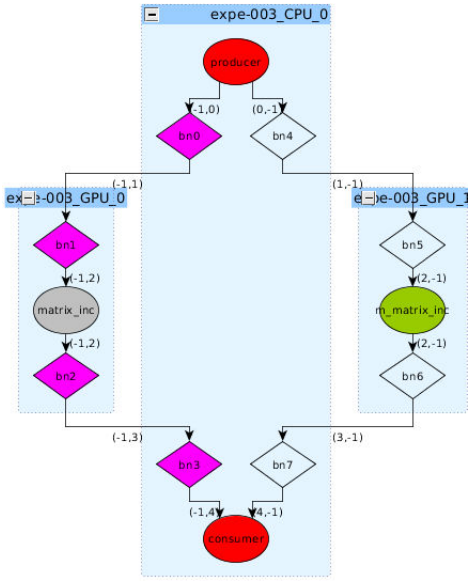
Durant la première étape, il faut allouer les ressources mémoires (cf. Figure 7.4). En effet, les buffers sont le support du flux des données. Sans buffer, il est impossible de rediriger un flux de données. Après création des ressources mémoire, si la tâche décédée est suspendue et que la tâche réssuscitée est lancée, les jetons de données transitant dans le pipeline de buffers entre la tâche précédente et la tâche suivante seront perdus. Ce point est traité dans la deuxième étape.

Dans la deuxième étape, on souhaite rediriger et réguler le flux des données entre le pipeline décédé et le pipeline ressuscité. La solution consiste à exécuter les deux pipelines simultanément : on redirige le flux de données vers le pipeline ressuscité pendant que le pipeline décédé, amené à disparaître, termine son flot d'exécution. Il faut toutefois s'assurer que les données sortant du pipeline décédé atteignent la tâche suivante avant les données du pipeline



(a) Latence écourtée

(b) Latence allongée



(c) Latence identique

FIGURE 7.5 – Etape 2. A la redirection du flux des données, trois types de scénarios apparaissent dépendants de la différence de longueur entre les chemins de pipelines décédé et ressuscité : la latence écourtée, la latence identique et la latence allongée.

ressuscité. Autrement, le flux des données sera désordonné. Pour traiter ce problème, trois scénarios sont possibles en fonction de la différence de latence entre le pipeline décédé et le pipeline ressuscité. La latence dont nous faisons mention ici est identique à celle que nous avons décrite au chapitre précédent. En fait, la différence de latence correspond tout simplement à la différence du nombre de buffers sur les deux pipelines. Ci-dessous la description de ces trois scénarios illustré également sur la Figure 7.5.

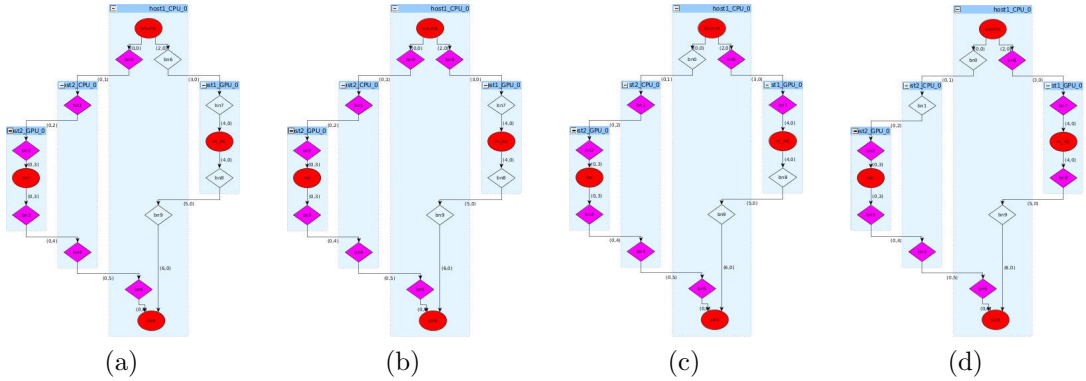
- Une latence écourtée : le chemin parcouru par le flux de données est plus court après qu'avant la migration.
- Une latence identique : les chemins sont de longueurs équivalentes.
- Une latence allongée : le chemin parcouru par le flux de données est plus long après qu'avant la migration.

Dans le cas d'une latence écourtée, le temps que les données du pipeline décédé soient complètement purgées, les données du pipeline ressuscité seront déjà disponibles. Il faut donc arrêté le flux des données sur ce pipeline, le temps que l'autre ait terminé de se vider. A noter que, dans ce scénario, le flux de données n'est pas perturbé. La Figure 7.6 présente le déroulement de l'opération de migration dans le cas d'une latence écourtée. Les buffers de couleur violette contiennent des données tandis que les transparents n'en contiennent pas. Ainsi, le flux de données est illustré par le flux des buffers de couleur violette. Dans le cas d'une latence identique, l'exécution simultanée des deux pipelines ne pose aucun problème et le flux des données n'est pas perturbé. Dans le cas d'une latence allongée, un phénomène de bulle (absence de données) se produit dans le reste du pipeline de l'application parce que le pipeline ressuscité est trop long. La latence est donc trop grande pour pouvoir enchaîner directement avec la terminaison du pipeline décédé.

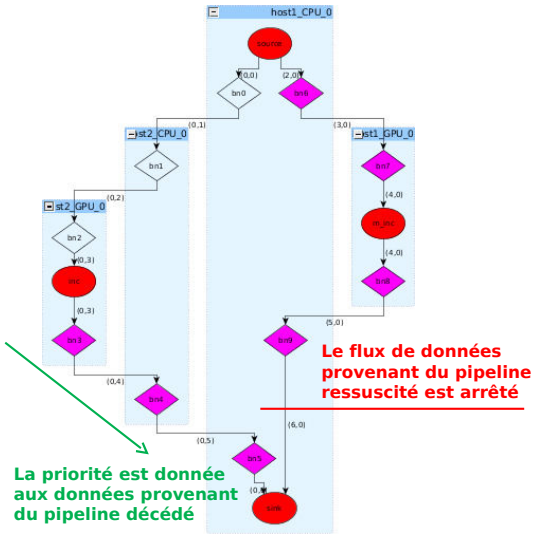
Nous avons abordé l'allocation des buffers ainsi que la redirection du flux des données dans le cas d'une tâche SANS état. Il reste désormais à traiter le cas de la migration d'une tâche AVEC état. Dans ce cas, chaque pipeline (décédé et ressuscité) est divisé en deux : la partie qui précède la tâche migrée et la partie qui suit la tâche migrée. Les deux pipelines fonctionnent simultanément jusqu'à ce que le flux de données de chaque pipeline atteigne la tâche migrée ; le pipeline qui atteint la tâche migrée en premier attend l'autre. Quand les deux pipelines sont synchronisés sur la tâche migrée. L'état interne de la tâche décédée est alors transmis à la tâche ressuscitée qui prendre le relais. Après cela, le fonctionnement des pipelines est similaire à la migration de tâche SANS état. La Figure 7.7 récapitule les étapes de la migration dans ce cas particulier.

7.3 Résultats

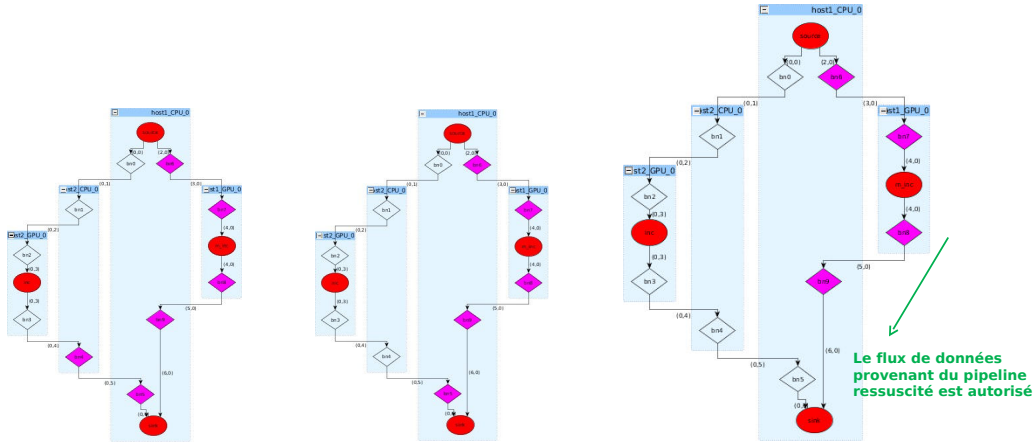
Les trois scénarios de latence de la migration de tâches ont été testés et validés sur la plateforme cluster du laboratoire. La Figure 7.5a présente une des applications test tandis que les résultats de la migration de tâche appliquée à cette application avec une tâche SANS état



(a) (b) (c) (d)

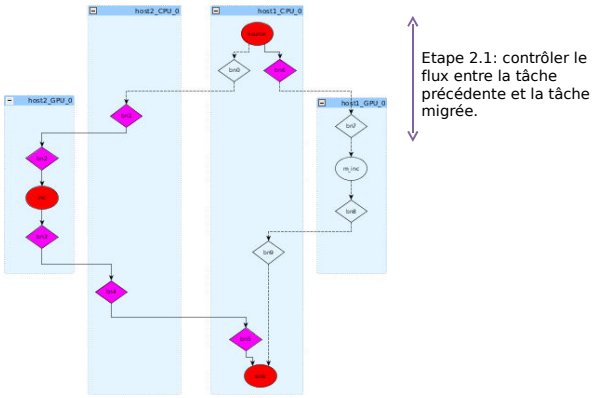


(e)

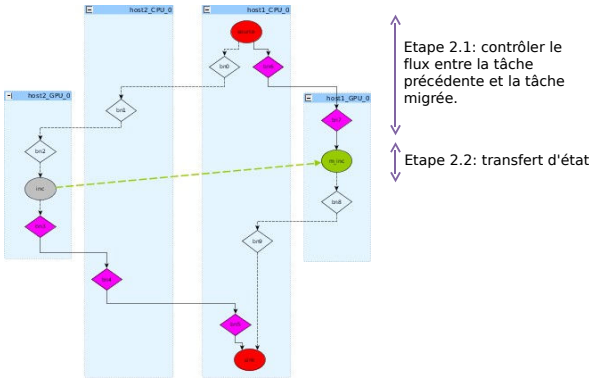


(f) (g) (h)

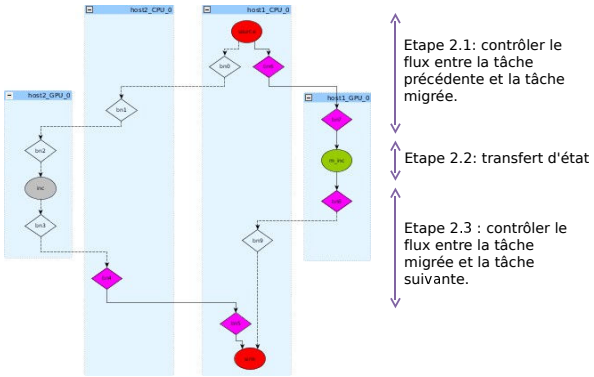
FIGURE 7.6 – Etape 2. Illustration du processus de redirection du flux de données dans le cas d'une latence écourtée lors de la migration d'une tâche SANS état. Les deux pipelines fonctionnent simultanément jusqu'à ce que le flux des données dans le pipeline réssuscité atteigne la tâche suivante. Alors, ce dernier est arrêté le temps que le pipeline décédé ait fini de purger ses données.



(a) Avant d'atteindre la tâche migrée.



(b) Synchronisation sur la tâche migrée et transfert de l'état interne.



(c) Après le transfert d'état.

FIGURE 7.7 – Etape 2. Illustration du processus de redirection du flux de données dans le cas d'une latence écourtée lors de la migration d'une tâche AVEC état. Les deux pipelines fonctionnent simultanément jusqu'à ce que le flux de données de chaque pipeline atteigne la tâche migrée ; le pipeline qui atteint la tâche migrée en premier attend l'autre. Quand les deux pipelines sont synchronisés sur la tâche migrée. L'état interne de la tâche décédée est alors stable et prêt à être transmis à la tâche ressuscitée qui prendre le relais. Après cela, le fonctionnement des pipelines est similaire à la migration de tâche SANS état.

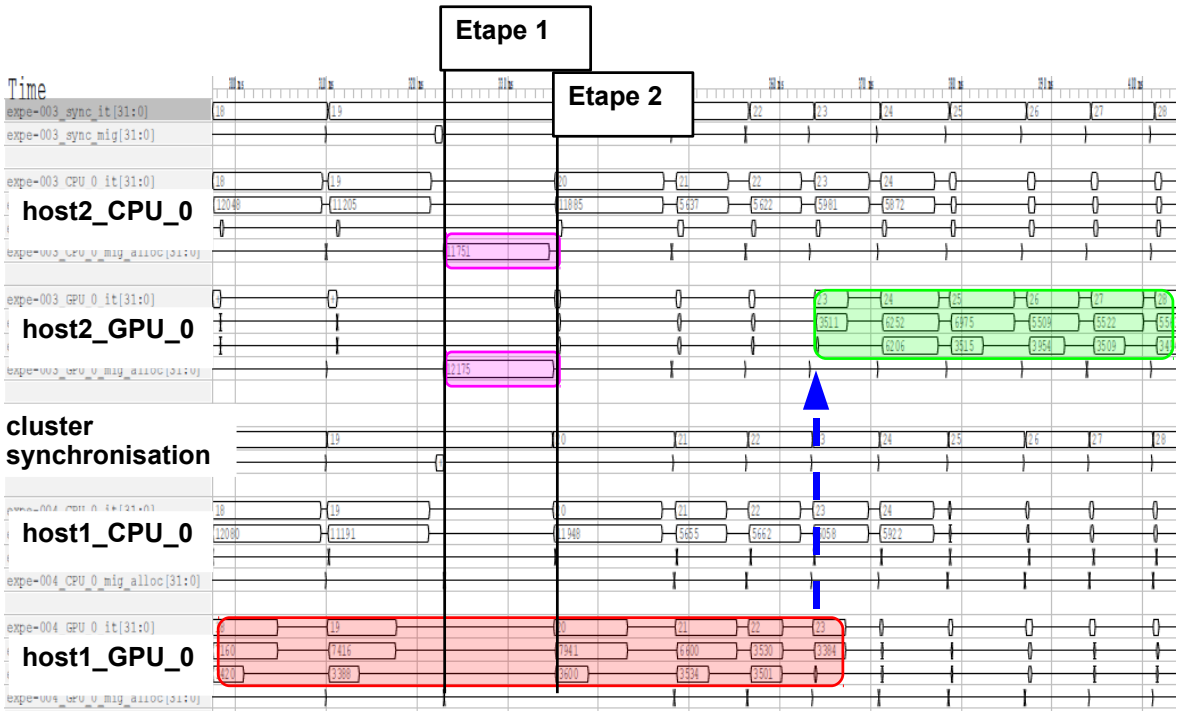


FIGURE 7.8 – Chronogramme résultat de migration de tâche SANS état opérée sur le cas illustré dans la Figure 7.5a. En rose, les allocations buffers de la première étape. L'étape 2 est ensuite enclenchée : la redirection des flux de données et la migration de la tâche décédée (en rouge) vers la tâche ressuscitée (en vert). Concernant les différentes lignes de chronogramme, il en existe 4 par élément de calcul : la première est un compteur d'itérations, la deuxième représente les transferts (CPU-CPU ou CPU-GPU), la troisième représente les temps de calcul du kernel (CPU ou GPU) et la dernière représente le temps d'allocation des buffers du pipeline ressuscité.

et une latence écourtée apparaissent dans le chronogramme de la Figure 7.8. Nous pouvons distinguer deux étapes. L'étape 1, représentant les allocations buffers, apparaît en rose. L'étape 2 est ensuite enclenchée avec la redirection des flux de données et la migration de la tâche décédée (en rouge) vers la tâche ressuscitée (en vert). Concernant les différentes lignes de chronogramme, il en existe 4 par élément de calcul : la première est un compteur d'itérations, la deuxième représente les transferts (CPU-CPU ou CPU-GPU), la troisième représente les temps de calcul du kernel (CPU ou GPU) et la dernière représente le temps d'allocation des buffers du pipeline ressuscité.

7.4 Conclusion

Une technique de migration a été incorporée à la librairie PACCOlib. La stratégie employée minimise les perturbations de la migration sur le fonctionnement de l'application. Cela est opéré grâce au découpage en plusieurs étapes de l'opération de migration et à son exécution sur plusieurs itérations. Le travail d'implémentation a été validé sur un cluster multi-GPU.

Concernant les perspectives, on cherchera à minimiser le temps d'allocation des buffers. Pour l'instant, cette allocation est effectuée en une seule fois sur tout le système et peut occuper suivant la taille des données une part assez conséquante du temps consommé durant une itération de l'application. Une allocation graduelle, en fonction de l'avancée du flux de données dans le pipeline ressuscité, peut être envisageable. Une autre solution consisterait à créer notre propre gestionnaire mémoire : un espace mémoire de taille supérieure à celle nécessaire pour les allocations initiales serait alloué à l'initialisation. Le gestionnaire mémoire gèrerait ensuite les futures allocations en fournissant des pointeurs vers les espaces mémoires libres. Dans le cas où, l'espace mémoire ne suffit plus un autre bloc mémoire serait alors alloué. Il sera alors nécessaire de se demander quelle taille de bloc mémoire le gestionnaire allouerait-il ? Serait-elle fixe ? Variable ? Comment choisir sa taille ?

Un autre axe d'amélioration est la migration simultanée de plusieurs tâches. Il faudra pour cela vérifier que des migrations multiples n'impliquent pas des recouvrements de communication trop encombrants qui pourraient congestionner le réseau.

Aussi, cette méthode est pour l'instant dénudée de toute intelligence. C'est au programmeur de décider du mapping et de la migration des tâches. Avec l'essor des plateformes de calcul HPC, que ce soit dans le publique ou dans l'entreprise, il pourrait être intéressant d'ajouter un algorithme d'ordonnancement dynamique des tâches. La question serait alors quelle stratégie d'ordonnancement adopter ? Quel fonction de coût et quels facteurs choisir pour décider de la migration d'un paramètre ? Cette méthode devra-t-elle être adoptée à des clusters de taille petite à moyenne ou à des clusters de taille plus imposante ?

Chapitre 8

Conclusion

Il y a 10 ans, il était inconcevable de faire le rapprochement entre un supercalculateur et un ordinateur grand public. Aujourd'hui, avec la démocratisation des CPU multi-processeur multi-coeur, l'utilisateur d'un ordinateur grand public détient entre ses mains une machine parallèle capable de produire une puissance de calcul relativement importante. Une tablette ou un smartphone dernière génération peut développer une puissance d'environ 1Gflop/s. Cela correspond à la puissance délivrée par le supercalculateur le moins puissant (sur la liste du top500) en 1995. C'était, il y a 18 ans ! De même, un ordinateur portable peut facilement développer une puissance de calcul de l'ordre de 70 Gflops. Cela correspond à la puissance délivrée par le supercalculateur le moins puissant en 2000. C'était, il y a 13 ans ! Alors bien entendu, on ne peut toujours pas comparer les performances d'un ordinateur grand public à celles d'un supercalculateur. Cependant, les ordinateurs grand public sont désormais constitués d'une architecture permettant le Calcul Haute Performance. Les solutions HPC, accélérant les calculs scientifiques, se sont rapidement répandues dans les milieux académiques et industriels. Cela s'est fait d'autant plus vite que des efforts considérables ont été entrepris au niveau des cartes graphiques pour améliorer le GPGPU. Il est néanmoins nécessaire, afin d'obtenir l'accélération attendue, que l'architecture exploitée le soit efficacement. Or, les modèles de programmation utilisés jusqu'à présent pour l'exploitation des architectures parallèles émergentes sont hérités des supercalculateurs. Une Adéquation Algorithme Architecture accompagnée de bonnes connaissances en architecture des processeurs est donc nécessaire.

Dans la première partie de cette thèse, nous montrons les performances atteignables sur une architecture HPC constituée d'un CPU connecté à un GPU. Un algorithme de traitement d'image 3D, basé sur des opérations de morphologie mathématiques, très utilisé en science des matériaux a été implémenté sur CPU et sur GPU. Il s'agit de la granulométrie. L'application a d'abord été optimisée en réduisant le nombre de bits nécessaires au stockage de l'information. Cette optimisation a permis de réduire l'espace mémoire alloué et d'augmenter le nombre de voxels traités par instruction. Nous avons vu que l'accélération, avec le standard de programmation parallèle OpenMP, de la version CPU du code de granulométrie est effectuée

de manière quasi-immédiate grâce aux annotations de code introduites par les directives pragma. Les performances ainsi obtenues sur le CPU seul se sont avérées très raisonnables en comparaison de l'investissement en temps qu'à nécessité l'implémentation sur GPU. Cette dernière implémentation apporte toutefois un gain en performance d'un ordre de magnitude supplémentaire comparé à la version CPU. Au final, le temps d'exécution de l'application, nécessitant originellement quelques heures de calcul, est passée à l'ordre de la minute sur CPU et à moins d'une minute sur GPU. Pour différentes tailles de volume, le résultat des mesures de performance sur CPU présente une courbe d'allure exponentielle. Par contre, le résultat des mesures de performance sur GPU affiche une allure qui, au premier coup d'oeil, ne correspond à aucun modèle mathématique. Un modèle de performance, permettant à partir d'une première mesure (sur une taille de volume réduite) d'extrapoler le temps d'exécution de l'application de granulométrie, a finalement été élaboré. Ce modèle est basé sur une formule mathématique prenant comme paramètres le temps d'exécution d'un bloc et le temps de chargement des blocs par le *warp scheduler*. Les autres paramètres sont calculés à partir de la liste des caractéristiques du GPU, accessibles à tout utilisateur. Ce modèle de performance, valable pour l'application de granulométrie, pourrait également s'étendre à d'autres algorithmes réguliers.

Dans la deuxième partie de la thèse, nous traitons des modèles de programmation pour architecture hétérogène. Afin de faciliter le portage d'applications sur GPU, il existe deux grandes façon de faire : les compilateurs "paralléliseurs" permettent de porter du code séquentiel (*legacy code*) vers des accélérateurs. Cependant, la programmation séquentielle n'est pas un modèle adapté à l'expression naturelle du parallélisme. Pour cela, la représentation par un *Data Flow Graph* est plus adaptée. Elle permet de décomposer naturellement l'application en tâches. L'organisation des tâches entre elles est assuré par leurs dépendances de données. Il est néanmoins nécessaire au programmeur de s'affairer à la programmation des communications ; ce qu'il n'était pas habituer à faire sur une architecture séquentielle. Cette tâche est d'autant plus complexe que les protocoles de communication ne sont pas tous unifiés dans un même standard. Le programmeur doit donc maîtriser différentes API ; principalement CUDA/OpenCL, les Pthreads et MPI dans le cas d'un cluster multi-GPU. Pour cela, nous avons développé, PACCOlib, une librairie permettant de simplifier la programmation par la génération automatique de code pour la gestion de la mémoire, des données et des threads. Nous avons choisi le modèle *Synchronous Data Flow* parce qu'il assure l'absence de *deadlocks* durant l'exécution et permet donc un ordonnancement des tâches. Afin de synchroniser le flux des données entre les éléments de calcul, nous avons choisi le modèle d'implémentation *Bulk Synchronous Parallel*. Ce modèle fonctionne pour une application itérative. Le système global est alors synchronisé à chaque itération : on lance les communications puis les calculs avant de synchroniser le système global. Un mode de fonctionnement permettant le recouvrement calculs-communications est également disponible. Le fonctionnement du système est alors réduit à deux étapes : les communications et les calculs simultanément puis la synchronisation globale. L'allocation des données, leurs transferts, la création et l'exécution des threads ainsi que la synchronisation globale du système sont gérés par notre outil. Bien que l'application soit exprimée avec un *Data Flow Graph*, le modèle d'implémentation *Bulk Synchronous Parallel* insère une synchronisation des données sur le flot d'instructions. Notre modèle n'est donc pas complètement orienté *Data Flow*. L'outil a été validé sur trois applications dont une

application streaming de calcul de carte de saillance visuelle. L'utilisation de notre outil a permis le déploiement efficace de l'application streaming sur plusieurs GPU. Pour une vidéo de dimension 720x576, le débit sortant de l'application a été significativement amélioré de 21 à 33 fps.

Tandis que le mapping des tâches du *Data Flow Graph* sur le Graphe d'Architecture est effectué par le programmeur, leur ordonnancement est effectué par l'outil à l'exécution. Cet ordonnancement est statique et ne peut plus être modifié en cours d'exécution. A supposer que le choix de mapping de l'utilisateur ne soit pas la meilleure répartition possible sur le cluster, le programmeur doit donc arrêter l'application pour rectifier son mapping. Afin que le programmeur puisse modifier l'emplacement des tâches au cours de l'exécution, une méthode de migration de tâches a été développée. Pour l'instant, elle permet de modifier l'emplacement d'une seule tâche à la fois. Celle-ci peut être migrée sur n'importe quel élément de calcul du cluster. A terme, nous souhaitons permettre la migration de plusieurs tâches à la fois.

Pour conclure, nous avons présenté une implémentation optimisée de l'algorithme de granulométrie sur CPU et GPU. Les gains apportés ont été considérables. Cependant, l'effort d'implémentation n'a pas été le même sur les deux architectures. Sur la GT200, le GPU nécessite une bonne maîtrise de la hiérarchie de ses espaces mémoires, une gestion manuelle des effets de bords (quand on utilise la mémoire partagée) et de suivre certaines règles d'accès pour obtenir des accès mémoires optimisés. Le CPU, par contre, peut fournir des performances importantes en parallélisant du code séquentiel avec de simples annotations. Avec le standard OpenACC, la parallélisation de code sur GPU deviendra peut être tout aussi simple que sur CPU. Cependant, sera-t-il possible d'organiser les données dans la hiérarchie mémoire afin d'obtenir de meilleures performances comme nous avons pu le faire manuellement ?

Quant à PACCOlib, ce modèle de programmation a été testé et validé avec des applications streaming sur cluster multi-GPU. Les perspectives vis à vis de ce modèle de programmation sont nombreuses. Plusieurs projets touchant à la programmation d'une architecture hétérogène existent. Parmi tous ces projets, PACCOlib pourrait en premier recours s'interfacer avec PREESM. PREESM est un outil, semblable à SynDEX, développé pour le prototypage rapide d'applications de traitement du signal et de l'image. Il s'applique au même domaine d'application que PACCOlib, présente une interface graphique et enrichi le mapping des tâches sur les éléments de calcul en procurant des informations supplémentaires contenues dans un "scénario". Cependant, il ne cible pas les GPU. En deuxième recours, notre modèle pourrait être étendu pour supporter d'autres types de parallélisme structuré : les squelettes algorithmiques [25], par exemple. Afin d'enrichir le mécanisme d'ordonnancement dynamique de notre outil, un support exécutif pourrait être emprunté (StarPU par exemple ?) ou mis en place : ajouter un/des algorithme(s) basés sur des fonctions de coûts (peut-être inspirés de KAAPI ?) pourraient permettre à l'outil de gérer les migrations de façon automatique.

Annexe A

Bases de morphologie mathématique

La Morphologie Mathématique (MM), fondée par Georges Matheron et Jean Serra [105], est une théorie et technique mathématique basée sur la théorie des ensembles, la théorie des treillis, la topologie et la probabilité.

Durant sa thèse, Jean Serra alors sous la direction de Georges Matheron, développa plusieurs opérations permettant d'extraire des informations d'une image. Le principe fut d'appliquer un traitement sur une image à partir d'une forme géométrique. Ainsi naquit la morphologie mathématique. A partir d'un nombre restreint d'opérations, une diversité de résultats peut être obtenus selon la forme géométrique du masque appliqué sur l'image. De plus, en appliquant successivement différentes opérations de MM, les combinaisons de résultat possibles deviennent très importantes. L'érosion et la dilation sont deux opérations rudimentaires en MM et pourtant elles permettent des applications de traitement d'images très appréciées. La granulométrie basée sur ces deux opérations en est un exemple probant. La MM fournit aussi des outils de filtrage, segmentation, quantification et modélisation d'images.

Ne pouvant pas effectuer un cours complet sur la MM, nous nous limiterons aux notions strictement nécessaires pour la compréhension de l'algorithme de granulométrie : la morphologie mathématique ensembliste. Cette dernière s'applique sur des images binaires et nous convient donc parfaitement. La morphologie mathématique fonctionnelle, à la différence de la morphologie mathématique ensembliste, s'applique aux images en niveaux de gris.

Une image binaire est généralement obtenue à partir d'une image en niveaux de gris (nvg), au moyen d'une technique de seuillage (Figure A.1). Chaque pixel d'une image binaire ne peut prendre que les deux valeurs zéro ou un. Cela représente un contexte simple permettant

une formalisation mathématique des problèmes par des outils tels que la topologie. Certaines applications (détection de défauts, contrôle qualité, métrologie) nécessitent le passage d'une image nvg en binaire.

(a) niveaux de gris (b) binaire

FIGURE A.1 – Exemple d'image en niveaux de gris et binaire (après seuillage).

L'image binaire renferme un certain nombre de régions (ensemble de pixels connexes) codées à 1 que l'on peut définir comme des objets d'intérêt, par rapport à un fond codé à 0. Une transformation morphologique utilise un ensemble particulier de centre x, de géométrie et de taille connues, appelé élément structurant. L'élément structurant est déplacé de façon à ce que son centre x passe successivement par toutes les positions possibles dans l'image binaire. Pour chacune des positions du centre de l'élément structurant, on se pose une question relative à l'union ou à l'intersection de l'élément structurant avec les objets de l'image (Figure A.2). L'ensemble des points correspondant à une réponse positive permet de construire une nouvelle image résultat.

Pour l'érosion (Figure A.3), la question posée est : l'élément structurant est-il complètement inclus dans la région de l'image ? (c'est-à-dire : l'intersection est-elle l'élément structurant lui-même ?)

Si l'élément structurant B déplacé au point x est inclus dans un objet de l'image, le point x est conservé dans la transformée. Les objets de taille inférieure à celle de l'élément structurant disparaissent de l'image résultat. Les autres objets sont "amputés" d'une partie correspondant à la taille de l'élément structurant. S'il existe des trous dans les objets, c'est à dire des "morceaux" de fond à l'intérieur des objets, ils sont accentués. Des parties d'un objet initialement reliées entre elles peuvent être séparées.

Pour la dilatation (Figure A.4), la question posée est : l'intersection entre l'élément structurant et la région de l'image est-elle non vide ?

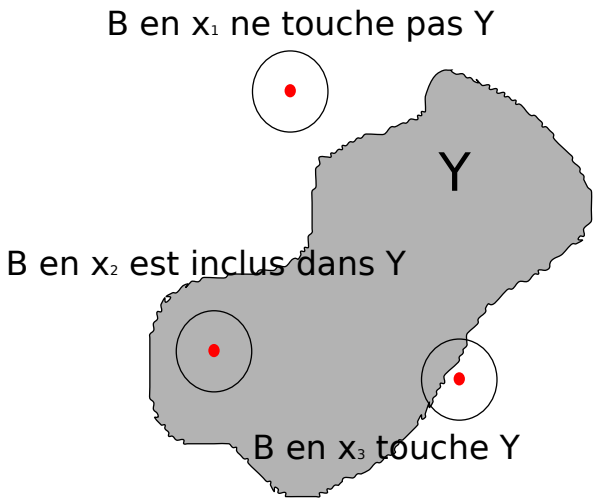


FIGURE A.2 – Union/intersection de l'élément structurant B avec l'objet Y de l'image.

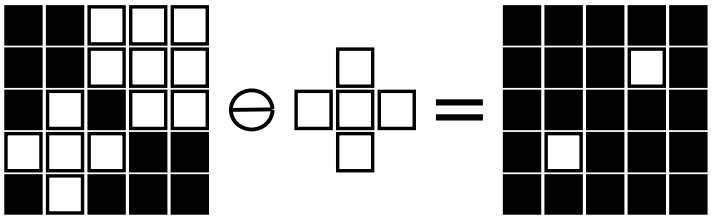


FIGURE A.3 – Erosion par un élément structurant en forme de croix.

Si l'élément structurant B déplacé au point x touche au moins un objet de l'image, le point x est conservé dans la transformée. La dilatation est l'opération duale (ou inverse) de l'érosion : elle consiste à éroder le complémentaire de l'image, puis à complémenter le résultat. Tous les objets "grossissent" d'une partie correspondant à la taille de l'élément structurant. S'il existe des trous dans les objets, c'estàdire des "morceaux" de fond à l'intérieur des objets, ils peuvent être comblés. Si des objets sont situés à une distance moins grande que la taille de l'élément structurant, il peuvent fusionner.

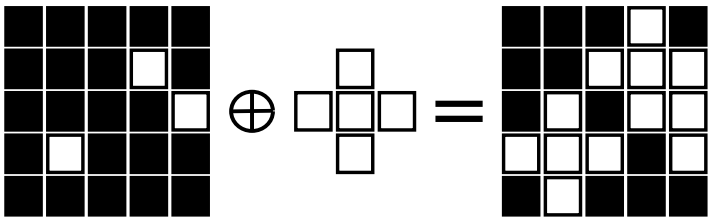
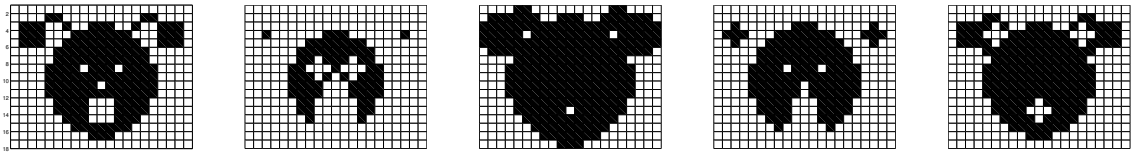


FIGURE A.4 – Dilatation par un élément structurant en forme de croix.

Une ouverture est la combinaison d'une érosion suivie d'une dilatation tandis qu'une fermeture est une dilatation suivie d'une érosion (Figure A.5).



(a) originale (b) érosion (c) dilatation (d) ouverture (e) fermeture

FIGURE A.5 – Résultat de différentes opérations de MM avec une croix comme élément structurant.

Annexe B

Caractéristiques des GPUs

B.1 Caractéristiques générales

Voici, dans l'ordre chronologique, la liste des GPUs sur lesquels le portage de l'algorithme de granulométrie a été étudié :

- GeForce 8600 GS
- GeForce 9500 GT
- GeForce GTX 285
- GeForce GTX 480
- Quadro 4000

Le Tableau B.1 résume les caractéristiques des trois principaux GPU qui ont servis de support d'étude.

B.2 Accès coalesced à la mémoire globale

	GeForce GTX 285	GeForce GTX 480	Quadro 4000
CUDA Driver / Runtime Version	4.0 / 4.0	4.0 / 4.0	4.0 / 4.0
CUDA Compute Capability	1.3	2.0	2.0
Total amount of global memory	2047 MBytes	1536 MBytes	2047 MBytes
Total number of CUDA cores	30 MP x 8 SP = 240 CUDA Cores	15 MP x 32 SP = 480 CUDA Cores	8 MP x 32 SP = 256 CUDA Cores
GPU Clock Speed	1.48 GHz	1.40 GHz	0.95 GHz
Memory Clock rate	1242.00 Mhz	1848.00 Mhz	1404.00 Mhz
Memory Bus Width	512-bit	384-bit	256-bit
L2 Cache Size	-	786432 bytes	524288 bytes
Max Texture Dimension Size (x,y,z)	1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(8192) x 512, 2D=(8192,8192) x 512	1D=(16384) x 2048, 2D=(16384,16384) x 2048	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory :	65536 bytes	65536 bytes	65536 bytes
Total amount of shared memory per block	16384 bytes	49152 bytes	49152 bytes
Total number of registers available per block	16384	32768	32768
Warp size	32	32	32
Maximum number of threads per block	512	1024	1024
Maximum sizes of each dimension of a block	512 x 512 x 64	1024 x 1024 x 64	1024 x 1024 x 64
Maximum sizes of each dimension of a grid	65535 x 65535 x 1	65535 x 65535 x 65535	65535 x 65535 x 65535
Maximum memory pitch	2147483647 bytes	2147483647 bytes	2147483647 bytes
Texture alignment	256 bytes	512 bytes	512 bytes
Concurrent copy and execution	Yes with 1 copy engine(s)	Yes with 1 copy engine(s)	Yes with 2 copy engine(s)
Run time limit on kernels	Yes	No	Yes
Integrated GPU sharing Host Memory	No	No	No
Support host page-locked memory mapping	Yes	Yes	Yes
Concurrent kernel execution	No	Yes	Yes
Alignment requirement for Surfaces	Yes	Yes	Yes
Device has ECC support enabled	No	No	No
Device is using TCC driver mode	No	No	No
Device supports Unified Addressing (UVA)	No	Yes	Yes
Device PCI Bus ID / PCI location ID	4 / 0	8 / 0	7 / 0
Compute Mode :	Default (multiple host threads can use cudaSetDevice() with device simultaneously)	Default (multiple host threads can use cudaSetDevice() with device simultaneously)	Default (multiple host threads can use cudaSetDevice() with device simultaneously)

TABLE B.1 – Caractéristiques des cartes GPUs.

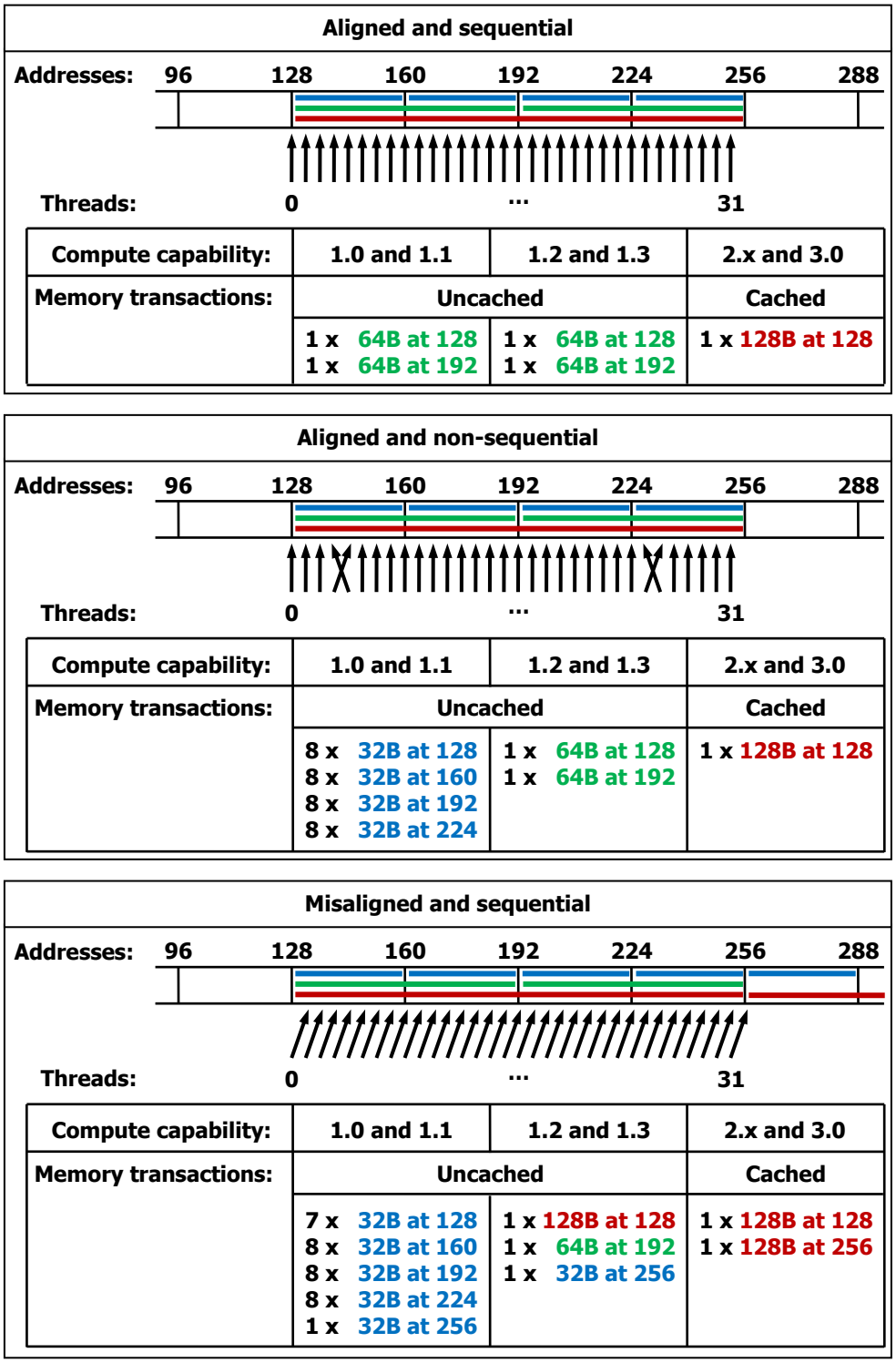


Figure F-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

Annexe C

Code des différentes versions de kernels testés avec le profiler

```
// TEST : correpond aux diff rentes impl mentations de kernels test s
// TEST VALIDE : correspond aux kernels qui sont utilis s dans la version actuelle de
     l'application de granulom trie

/*=============================================================================
                              MACROS
=============================================================================*/

#define CURRENT current_no_borders[threadIdx.y*(blockDim.x+2)+threadIdx.x]

#define UP       current_no_borders[(threadIdx.y-1)*(blockDim.x+2)+threadIdx.x]
#define DOWN     current_no_borders[(threadIdx.y+1)*(blockDim.x+2)+threadIdx.x]
#define NEXT     current_no_borders[threadIdx.y*(blockDim.x+2)+threadIdx.x+1]
#define PREVIOUS current_no_borders[threadIdx.y*(blockDim.x+2)+threadIdx.x-1]

#define TEMP            temp[threadIdx.y*blockDim.x+threadIdx.x]
#define BIT_COUNTER     bit_counter[threadIdx.y*blockDim.x+threadIdx.x]

/*=============================================================================
                              FUNCTIONS
=============================================================================*/

/****************** Cas d'une image 2D ************************/

// TEST des acc s m moire en 1D
__global__ void erosion1D_gmem(unsigned int *odata, unsigned int *idata, const int
    dimx, const int dimy)
{
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;

        if (xindex >= dimx*dimy)
                return;

        odata[xindex] = idata[xindex]
                              & idata[xindex-dimx]
                              & idata[xindex+dimx]
                              & ((idata[xindex] >> 1) | (idata[xindex+1] << 31))
                              &  ((idata[xindex] << 1) | (idata[xindex-1] >> 31));
```

```
}

// TEST de diff rence de temps de calcul entre kernel  rosion  (op rateur ET) et
    dilatation (op rateur OU)
__global__ void dilatation1D_gmem(unsigned int *odata, unsigned int *idata, const int
    dimx, const int dimy)
{
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;

        if (xindex >= dimx*dimy)
                return;

        odata[xindex] = idata[xindex]
                                | idata[xindex-dimx]
                                | idata[xindex+dimx]
                                | ((idata[xindex] >> 1) | (idata[xindex+1] << 31))
                                |  ((idata[xindex] << 1) | (idata[xindex-1] >> 31));
}

// TEST de diff rence de temps de calcul entre d coupage des blocs en 1D et en 2D
__global__ void dilatation2D_gmem(unsigned int *odata, unsigned int *idata, size_t
    pitch, const int dimx, const int dimy)
{
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;

        if (xindex >= dimx && yindex >= dimy)
                return;

        odata[xindex + yindex*pitch] = idata[xindex + yindex*pitch]
                                             | idata[xindex + (yindex-1)*pitch]
                                             | idata[xindex + (yindex+1)*pitch]
                                             | (idata[xindex + yindex*pitch]  >> 1)
                                                 | (idata[xindex+1 + yindex*pitch]
                                                  << 31)
                                             | (idata[xindex + yindex*pitch] << 1)
                                                 | (idata[xindex-1 + yindex*pitch]
                                                >> 31);
}

// TEST de diff rence de temps de calcul avec/sans utilisation de la m moire
    partag e
__global__ void dilatation2D_shmem_outsidebloc(unsigned int *odata, unsigned int *
    idata, size_t pitch, const int dimx, const int dimy)
{
        extern __shared__ unsigned int current_tab[];
        unsigned int* current_no_borders = &current_tab[blockDim.x+3];
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;

        if (xindex >= dimx && yindex >= dimy)
                return;

        CURRENT = idata[xindex + yindex * pitch];

        //load borders at current slice for each block
        //left border
        if (threadIdx.x == 0)
                PREVIOUS = idata[(xindex-1) + yindex * pitch];
        //right border
        if (threadIdx.x == blockDim.x-1)
                NEXT = idata[(xindex+1) + yindex * pitch];
        //upper border
        if (threadIdx.y == 0)
```

```
                UP = idata[xindex + (yindex-1) * pitch];
        //lower border
        if (threadIdx.y == blockDim.y-1)
                        DOWN = idata[xindex + (yindex+1) * pitch];

        odata[xindex + yindex*pitch] = CURRENT | UP | DOWN | (CURRENT >> 1) | (NEXT <<
             31) | (CURRENT << 1) | (PREVIOUS >> 31);
}

// TEST de diff rence de temps de calcul d'utilisation de la m moire partag e avec
    le bloc de threads dont la taille est  gale     celle du bloc d coup  ou
    celle de la taille du bloc + le voisinage de voxels n cessaires pour les effets
    de bord
__global__ void dilatation2D_shmem_insidebloc(unsigned int *odata, unsigned int *idata
    , size_t pitch, const int dimx, const int dimy)
{
        extern __shared__ unsigned int current_tab[];
        unsigned int* current_no_borders = &current_tab[blockDim.x];
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;

        if (xindex >= dimx && yindex >= dimy)
                return;

        current_no_borders[threadIdx.y*blockDim.x+threadIdx.x] = idata[xindex + yindex
             * pitch];
        current_no_borders[-blockDim.x+threadIdx.x] = idata[xindex + (blockIdx.y*
            blockDim.y-1) * pitch]; // UP
        current_no_borders[blockDim.y*blockDim.x+threadIdx.x] = idata[xindex + (
            blockIdx.y*blockDim.y+blockDim.y) * pitch]; // DOWN

        if (threadIdx.x == 0 || threadIdx.x == blockDim.x)
                return;

        odata[xindex + yindex*pitch] = current_no_borders[threadIdx.y*blockDim.x+
            threadIdx.x]
                                        | current_no_borders[-blockDim.x+threadIdx.x]
                                        | current_no_borders[blockDim.y*blockDim.x+
                                            threadIdx.x]
                                        | (current_no_borders[threadIdx.y*blockDim.x+
                                            threadIdx.x] >> 1)
                                        | (current_no_borders[threadIdx.y*blockDim.x+
                                            threadIdx.x+1] << 31)
                                        | (current_no_borders[threadIdx.y*blockDim.x+
                                            threadIdx.x] << 1)
                                        | (current_no_borders[threadIdx.y*blockDim.x+
                                            threadIdx.x-1] >> 31);
}

/***************** Cas d'une image 3D ***********************/

// TEST du traitement d'image 3D en m moire globale
__global__ void dilatation3D_gmem(unsigned int *odata, unsigned int *idata, size_t
    pitch, const int dimx, const int dimy, const int dimz)
{
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;
        int slicePitch = pitch * dimy;
        int z = 0;
        unsigned int temp;

        if (xindex >= dimx && yindex >= dimy)
                return;
```

```
        for (z = 1; z < dimz-1; z++)
        {
                temp = idata[xindex + yindex * pitch + z * slicePitch];;
                temp |= (temp >> 1) | (temp << 1); // LEFT | RIGHT
                temp |= idata[xindex + (yindex-1) * pitch + z * slicePitch]; // UP
                temp |= idata[xindex + (yindex+1) * pitch + z * slicePitch]; // DOWN
                temp |= idata[xindex + yindex*pitch + (z+1)*slicePitch]; // IN
                temp |= idata[xindex + yindex * pitch + (z-1)*slicePitch]; // OUT

                if (xindex !=0)
                        temp |= (idata[(xindex-1) + yindex * pitch + z*slicePitch] >>
                            31);
                if (xindex != dimx-1)
                        temp |= (idata[(xindex+1) + yindex * pitch + z*slicePitch] <<
                            31);

                odata[xindex + yindex*pitch + z * slicePitch] = temp;
        }
}

// TEST VALIDE du traitement d'image 3D en m moire partag e
// C'est un des kernels qui a   t   choisi pour  tre  utilis  dans la version
    actuelle de l'application de granulom trie
__global__ void dilatation3D_shmem(unsigned int *odata, unsigned int *idata, size_t
    pitch, const int dimx, const int dimy, const int dimz)
{
        extern __shared__ unsigned int current_tab[];

        /*****************************************************/

        unsigned int* current_no_borders = &current_tab[blockDim.x+3];
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;
        int slicePitch = pitch * dimy;
        int z = 0;
        unsigned int in, out;

        // start erosion
        if (xindex >= dimx && yindex >= dimy)
                return;

        // first slice fetch in gmem
        CURRENT = idata[xindex + yindex * pitch];
        __syncthreads();
        out = CURRENT;

        // loop on the z-dimension
        for (; z < dimz; z++)
        {
                //load borders at current slice for each block
                //left border
                if (xindex ==0)
                        PREVIOUS = CURRENT << 31;
                else
                        if (threadIdx.x == 0)
                                PREVIOUS = idata[(xindex-1) + yindex * pitch + z*
                                    slicePitch];
                //right border
                if (xindex == dimx-1)
                        NEXT = CURRENT >> 31;
                else
                        if (threadIdx.x == blockDim.x-1)
                                NEXT = idata[(xindex+1) + yindex * pitch + z*
                                    slicePitch];
```

```
                //upper border
                if (yindex ==0)
                        UP = CURRENT;
                else if (threadIdx.y == 0)
                        UP = idata[xindex + (yindex-1) * pitch + z*slicePitch];
                //lower border
                if (yindex == dimy-1)
                        DOWN = CURRENT;
                else
                        if (threadIdx.y == blockDim.y-1)
                                DOWN = idata[xindex + (yindex+1) * pitch + z*
                                    slicePitch];

                if (z != dimz-1)
                        in = idata[xindex + yindex*pitch + (z+1)*slicePitch];
                else
        /*****************************************************/
                        in = 0x00000000;
                __syncthreads();

                odata[xindex + yindex*pitch + z * slicePitch] = CURRENT | UP | DOWN |
                    (CURRENT >> 1) | (NEXT << 31) | (CURRENT << 1) | (PREVIOUS >> 31)
                    | in | out;
                __syncthreads();

        /*****************************************************/
                out = CURRENT;
                __syncthreads();
                CURRENT = in;
                __syncthreads();
        }
        /*****************************************************/
}

// TEST VALIDE du traitement d'image 3D en m moire partag e avec le compteur de
    voxels d'int r t
// C'est un des kernels qui a t choisi pour tre utilis dans la version
    actuelle de l'application de granulom trie
__global__ void dilatation3D_shmem_counter(unsigned int *odata, unsigned int *idata,
    size_t pitch, const int dimx, const int dimy, const int dimz, unsigned int*
    device_counter)
{
        // set allocated shared memory
        extern __shared__ unsigned int shared_mem[];
        unsigned int *bit_counter = &shared_mem[0];
        unsigned int *temp = &shared_mem[blockDim.x*blockDim.y];
        unsigned int *current_tab = &shared_mem[blockDim.x*blockDim.y*2];

        // bit counter parameters
        unsigned int tid = threadIdx.x + blockDim.x*threadIdx.y;
        TEMP = 0;
        BIT_COUNTER = 0;
        __syncthreads();

        /*****************************************************/

        unsigned int* current_no_borders = &current_tab[blockDim.x+3];
        int xindex = blockIdx.x*blockDim.x + threadIdx.x;
        int yindex = blockIdx.y*blockDim.y + threadIdx.y;
        int slicePitch = pitch * dimy;
        int z = 0;
        unsigned int in, out;

        // start erosion
```

```
        if (xindex < dimx && yindex < dimy)
        {
                // first slice fetch in gmem
                CURRENT = idata[xindex + yindex * pitch];
                __syncthreads();
                out = CURRENT;

                // loop on the z-dimension
                for (; z < dimz; z++)
                {
                        //load borders at current slice for each block
                        //left border
                        if (xindex ==0)
                                PREVIOUS = CURRENT << 31;
                        else
                                if (threadIdx.x == 0)
                                        PREVIOUS = idata[(xindex-1) + yindex * pitch +
                                             z*slicePitch];
                        //right border
                        if (xindex == dimx-1)
                                NEXT = CURRENT >> 31;
                        else
                                if (threadIdx.x == blockDim.x-1)
                                        NEXT = idata[(xindex+1) + yindex * pitch + z*
                                             slicePitch];
                        //upper border
                        if (yindex ==0)
                                UP = CURRENT;
                        else if (threadIdx.y == 0)
                                UP = idata[xindex + (yindex-1) * pitch + z*slicePitch
                                     ];
                        //lower border
                        if (yindex == dimy-1)
                                DOWN = CURRENT;
                        else
                                if (threadIdx.y == blockDim.y-1)
                                        DOWN = idata[xindex + (yindex+1) * pitch + z*
                                             slicePitch];

                        if (z != dimz-1)
                                in = idata[xindex + yindex*pitch + (z+1)*slicePitch];
                        else
        /*************************************************/
                                in = 0x00000000;
                        __syncthreads();

                        TEMP = CURRENT | UP | DOWN | (CURRENT >> 1) | (NEXT << 31) | (
                            CURRENT << 1) | (PREVIOUS >> 31) | in | out;
                        __syncthreads();
                        odata[xindex + yindex*pitch + z * slicePitch] = TEMP;
                        BIT_COUNTER += __popc(TEMP);
                        __syncthreads();

        /*************************************************/
                        out = CURRENT;
                        __syncthreads();
                        CURRENT = in;
                        __syncthreads();
                }
        }
        /*************************************************/

        //count number of bits set to 1
if (blockDim.x*blockDim.y >= 1024)
```

```
        if (tid < 512) { bit_counter[tid] += bit_counter[tid + 512]; } __syncthreads()
            ;
if (blockDim.x*blockDim.y >= 512)
        if (tid < 256) { bit_counter[tid] += bit_counter[tid + 256]; } __syncthreads()
            ;
if (blockDim.x*blockDim.y >= 256)
        if (tid < 128) { bit_counter[tid] += bit_counter[tid + 128]; } __syncthreads()
            ;
if (blockDim.x*blockDim.y >= 128)
        if (tid <  64) { bit_counter[tid] += bit_counter[tid +  64]; } __syncthreads()
            ;

        if (tid < 32)
        {
                bit_counter[tid] += bit_counter[tid + 32]; __syncthreads();
                bit_counter[tid] += bit_counter[tid + 16]; __syncthreads();
                bit_counter[tid] += bit_counter[tid +  8]; __syncthreads();
                bit_counter[tid] += bit_counter[tid +  4]; __syncthreads();
                bit_counter[tid] += bit_counter[tid +  2]; __syncthreads();
                bit_counter[tid] += bit_counter[tid +  1]; __syncthreads();
        }

        if (tid == 0)
                atomicAdd(device_counter, bit_counter[tid]);
}
```

Bibliographie

[1] *Données sur les processeurs Intel*, http://www.intel.com/pressroom/kits/quickreffam.htm.

[2] *imorph*.

[3] *Intel's cilk website*.

[4] *Openmp standard*.

[5] *Posix.1c, threads extensions (ieee std 1003.1c-1995)*.

[6] *The design and implementation of a first-generation CELL processor*, 2005.

[7] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati, *Targeting heterogeneous architectures via macro data flow*, Intl. Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU), HiPEAC, January 2012, pp. 1–6.

[8] Advanced Micro Devices Corporation (AMD), *Amd stream computing user guide*, december 2008.

[9] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, *Data-Aware task scheduling on multi-accelerator based platforms*, 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, December 2010, pp. 291–298.

[10] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí, *A proposal to extend the openmp tasking model for heterogeneous architectures*, Proceedings of the 5th International Workshop on OpenMP : Evolving OpenMP in an Age of Extreme Parallelism (Berlin, Heidelberg), IWOMP '09, Springer-Verlag, 2009, pp. 154–167.

[11] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí, *An extension of the starss programming model for platforms with multiple gpus*, Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Berlin, Heidelberg), Euro-Par '09, Springer-Verlag, 2009, pp. 851–862.

[12] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta, *Extending openmp to survive the heterogeneous multi-core era*, International Journal of Parallel Programming (2010), 440–459.

[13] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí, *Parallelizing dense and banded linear algebra libraries using smpss*, Concurr. Comput. : Pract. Exper. **21** (2009), no. 18, 2438–2456.

[14] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta, *Cellss : a programming model for the cell be architecture*, ACM/IEEE CONFERENCE ON SUPERCOMPUTING, ACM, 2006, p. 86.

[15] Siegfried Benkner, Enes Bajrovic, Erich Marth, Martin Sandrieser, Raymond Namyst, and Samuel Thibault, *High-level support for pipeline parallelism on many-core architectures*, Europar - International European Conference on Parallel and Distributed Computing - 2012 (Rhodes Island, Grèce), August 2012 (Anglais).

[16] Guy E. Blelloch, *Nesl : A nested data-parallel language (version 2.6)*, Tech. report, Pittsburgh, PA, USA, 1993.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, *Cilk : An efficient multithreaded runtime system*, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (Santa Barbara, California), July 1995, pp. 207–216.

[18] Robert D. Blumofe and Charles E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM **46** (1999), no. 5, 720–748.

[19] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli, *State-of-the-art in heterogeneous computing*, Sci. Program. **18** (2010), no. 1, 1–33.

[20] Emmanuel Brun, *De l'imagerie 3d des structures à l'étude des mécanismes de transport en milieux cellulaires*, Ph.D. thesis, AIX-MARSEILLE UNIVERSITE, September 2009.

[21] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan, *Brook for gpus : stream computing on graphics hardware*, ACM Trans. Graph. **23** (2004), no. 3, 777–786.

[22] David R. Butenhof, *Programming with posix threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[23] Daniel Castaño-Díez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S Frangakis, *Performance evaluation of image processing algorithms on the gpu.*, Journal of Structural Biology **164** (2008), no. 1, 153–160.

[24] B. B. Chaudhuri, *An efficient algorithm for running window pel gray level ranking 2-d images*, Pattern Recogn. Lett. **11** (1990), no. 2, 77–80.

[25] Murray Cole, *Algorithmic skeletons : structured management of parallel computation*, MIT Press, Cambridge, MA, USA, 1991.

[26] M. Coster and J. L. Chermant, *Precis d' analyse d'image*, Presses du CNRS, 1985,.

[27] Jr. Sterling J. Crabtree, Li-Ping Yuan, and Robert Ehrlich, *A fast and accurate erosion-dilation method suitable for microcomputers*, CVGIP : Graph. Models Image Process. **53** (1991), no. 3, 283–290.

[28] L. Dagum and R. Menon, *OpenMP : an industry standard API for shared-memory programming*, IEEE Computational Science and Engineering **5** (1998), no. 1, 46–55.

[29] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter, *The scalable heterogeneous computing (shoc) benchmark suite*, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (New York, NY, USA), GPGPU '10, ACM, 2010, pp. 63–74.

[30] Usman Dastgeer, *Skeleton programming for heterogeneous gpu-based systems*, 2011, p. 90.

[31] Gregory F. Diamos and Sudhakar Yalamanchili, *Harmony : an execution model and runtime for heterogeneous many core systems*, Proceedings of the 17th international symposium on High performance distributed computing (New York, NY, USA), HPDC '08, ACM, 2008, pp. 197–200.

[32] R. Dolbeau, S. Bihan, and F. Bodin, *Hmpp : A hybrid multi-core parallel programming environment*, Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[33] Jack Dongarra and Piotr Luszczek, *Linpack benchmark*, Encyclopedia of Parallel Computing, 2011, pp. 1033–1036.

[34] Ralf Ebner and Alexander Pfaffinger, *Transformation of functional programs into data flow graphs implemented with pvm*, Proceedings of the Third European PVM Conference on Parallel Virtual Machine (London, UK, UK), EuroPVM '96, Springer-Verlag, 1996, pp. 251–258.

[35] Johan Enmyren and Christoph W. Kessler, *Skepu : a multi-backend skeleton programming library for multi-gpu systems*, Proceedings of the fourth international workshop on High-level parallel programming and applications (New York, NY, USA), HLPP '10, ACM, 2010, pp. 5–14.

[36] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover, *GPU cluster for high performance computing*, Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference, IEEE, November 2004, pp. 47– 47.

[37] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan, *Sequoia : Programming the memory hierarchy*, Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006.

[38] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa Badia, Eduard Ayguade, and Jesus Labarta, *Optimizing the exploitation of multicore processors and gpus with openmp and opencl*, Languages and Compilers for Parallel Computing (Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, eds.), Lecture Notes in Computer Science, vol. 6548, Springer Berlin / Heidelberg, 2011, 10.1007/978-3-642-19595-2_15, pp. 215–229.

[39] O. Fluck, C. Vetter, W. Wein, A. Kamen, B. Preim, and R. Westermann, *A survey of medical image registration on graphics hardware*, Comput. Methods Prog. Biomed. **104** (2011), no. 3, e45–e57.

[40] J. Fung and S. Mann, *Using graphics devices in reverse : Gpu-based image processing and computer vision*, Multimedia and Expo, 2008 IEEE International Conference on, 23 2008-april 26 2008, pp. 9 –12.

[41] Nicolas GAC, *Adéquation algorithme architecture pour la reconstruction 3d en imagerie médicale tep*, Ph.D. thesis, Institut polytechnique de Grenoble, 17 juillet 2008.

[42] Nicolas Gac, StéPhane Mancini, Michel Desvignes, and Dominique Houzet, *High speed 3d tomography on cpu, gpu, and fpga*, EURASIP J. Embedded Syst. **2008** (2008), 5 :1–5 :12.

[43] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille, *Athapascan-1 : On-line building data flow graph in a parallel language*, Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '98, IEEE Computer Society, 1998, pp. 88–.

[44] Thierry Gautier, Xavier Besseron, and Laurent Pigeon, *Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors*, Proceedings of the 2007 international workshop on Parallel symbolic computation (New York, NY, USA), PASCO '07, ACM, 2007, pp. 15–23.

[45] Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin, *X-Kaapi : a Multi Paradigm Runtime for Multicore Architectures*, Rapport de recherche RR-8058, INRIA, feb 2012.

[46] Naga K. Govindaraju and Dinesh Manocha, *Cache-efficient numerical algorithms using graphics hardware*, Parallel Comput. **33** (2007), no. 10-11, 663–684.

[47] R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal **45** (1966), 1563–1581.

[48] T. Grandpierre, C. Lavarenne, and Y. Sorel, *Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors*, Proceedings of the seventh international workshop on Hardware/software codesign (New York, NY, USA), CODES '99, ACM, 1999, pp. 74–78.

[49] Tianyi David Han and Tarek S. Abdelrahman, *hicuda : a high-level directive-based language for gpu programming*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, pp. 52–61.

[50] R.M. Haralick, S. Chen, and T. Kanungo, *Recursive opening transform*, Computer Vision and Pattern Recognition, 1992. Proceedings CVPR '92., 1992 IEEE Computer Society Conference on, jun 1992, pp. 560–565.

[51] Robert M. Haralick, Stanley R. Sternberg, and Xinhua Zhuang, *Image analysis using mathematical morphology*, Pattern Analysis and Machine Intelligence, IEEE Transactions on **PAMI-9** (1987), no. 4, 532–550.

[52] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang, *Mars : a mapreduce framework on graphics processors*, Proceedings of the 17th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '08, ACM, 2008, pp. 260–269.

[53] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard, *Multi-gpu and multi-cpu parallelization for interactive physics simulations*, Proceedings of the 16th international Euro-Par conference on Parallel processing : Part II (Berlin, Heidelberg), Euro-Par'10, Springer-Verlag, 2010, pp. 235–246.

[54] Oscar Hernandez, Wei Ding, Barbara Chapman, Christos Kartsaklis, Ramanan Sankaran, and Richard Graham, *Experiences with high-level programming directives for porting applications to gpus*, Facing the Multicore - Challenge II (Rainer Keller, David Kramer, and Jan-Philipp Weiss, eds.), Lecture Notes in Computer Science, vol. 7174, Springer Berlin Heidelberg, 2012, pp. 96–107.

[55] Shrinidhi Hudli, Shrihari Hudli, Raghu Hudli, Yashonath Subramanian, and T. S. Mohan, *Gpgpu-based parallel computation : application to molecular dynamics problems*, Proceedings of the Fourth Annual ACM Bangalore Conference (New York, NY, USA), COMPUTE '11, ACM, 2011, pp. 10 :1–10 :8.

[56] Laurent Itti, Christof Koch, and Ernst Niebur, *A model of saliency-based visual attention for rapid scene analysis*, IEEE Trans. Pattern Anal. Mach. Intell. **20** (1998), 1254–1259.

[57] Ni Jiang, Wanneng Yang, Lingfeng Duan, Xiaochun Xu, Chenglong Huang, and Qian Liu, *Acceleration of ct reconstruction for wheat tiller inspection based on adaptive minimum enclosing rectangle*, Comput. Electron. Agric. **85** (2012), 123–133.

[58] L. V. Kale and Sanjeev Krishnan, *Charm++ : Parallel programming with message-driven objects*, Parallel Programming using C++ (Gregory V. Wilson and Paul Lu, eds.), MIT Press, 1996, pp. 175–213.

[59] Pavel Karas, Vincent Morard, Jan Bartovský, Thierry Grandpierre, Eva Dokládalová, Petr Matula, and Petr Dokládal, *Gpu implementation of linear morphological openings with arbitrary angle*, Journal of Real-Time Image Processing (2012), 1–15 (English).

[60] Kamran Karimi, Neil G. Dickson, and Firas Hamze, *A performance comparison of cuda and opencl*, CoRR **abs/1005.2581** (2010).

[61] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee, *Snucl : an opencl framework for heterogeneous cpu/gpu clusters*, Proceedings of the 26th ACM international conference on Supercomputing (New York, NY, USA), ICS '12, ACM, 2012, pp. 341–352.

[62] Jingfei Kong, Martin Dimitrov, Yi Yang, Janaka Liyanage, Lin Cao, Jacob Staples, Mike Mantor, and Huiyang Zhou, *Accelerating matlab image processing toolbox functions on gpus*, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (New York, NY, USA), GPGPU '10, ACM, 2010, pp. 75–85.

[63] David Kunzman, Gengbin Zheng, Eric Bohm, and Laxmikant V. Kalé, *Charm++, Offload API, and the Cell Processor*, Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism (Seattle, WA, USA), September 2006.

[64] Lucas Lattari, Anselmo Montenegro, Aura Conci, Esteban Clua, Virginia Mota, Marcelo Bernardes Vieira, and Gabriel Lizarraga, *Using graph cuts in gpus for color based human skin segmentation*, Integr. Comput.-Aided Eng. **18** (2011), no. 1, 41–59.

[65] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W. Toga, *Cuda optimization strategies for compute- and memory-bound neuroimaging algorithms*, Comput. Methods Prog. Biomed. **106** (2012), no. 3, 175–187.

[66] Edward A. Lee and David G. Messerschmitt, *Synchronous data flow : Describing signal processing algorithm for parallel computation*, COMPCON, 1987, pp. 310–315.

[67] Edward A. Lee and Thomas Parks, *Dataflow process networks*, Proceedings of the IEEE, 1995, pp. 773–799.

[68] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey, *Debunking the 100x gpu vs. cpu myth : an evaluation of throughput computing on cpu and gpu*, SIGARCH Comput. Archit. News **38** (2010), no. 3, 451–460.

[69] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng, *Merge : a programming model for heterogeneous multi-core systems*, Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ASPLOS XIII, ACM, 2008, pp. 287–296.

[70] Luke, Pascal Vallotton, and Dadong Wang, *Parallel van herk/gil-werman image morphology on gpus using cuda*, 2009.

[71] Sophie Marat, Tien Ho Phuoc, Lionel Granjon, Nathalie Guyader, Denis Pellerin, and Anne Guérin-Dugué, *Modelling spatio-temporal saliency to predict gaze direction for short videos*, Int. J. Comput. Vision **82** (2009), 231–243.

[72] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero, *Overlapping communication and computation by using a hybrid mpi/smpss approach*, Proceedings of the 24th ACM International Conference on Supercomputing (New York, NY, USA), ICS '10, ACM, 2010, pp. 5–16.

[73] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard, *Cg : a system for programming graphics hardware in a c-like language*, ACM Trans. Graph. **22** (2003), no. 3, 896–907.

[74] G. MATHERON, *Elements pour une théorie des milieux poreux*, Masson, Paris, 1967.

[75] G. Matheron, *Éléments pour une théorie des milieux poreux.*, Masson et Cie, Paris, 1967.

[76] R. Membarth, F. Hannig, J. Teich, M. Korner, and W. Eckert, *Frameworks for GPU accelerators : A comprehensive evaluation using 2D/3D image registration*, 2011 IEEE 9th Symposium on Application Specific Processors (SASP), IEEE, June 2011, pp. 78–81.

[77] Paulius Micikevicius, *3d finite difference computation on gpus using cuda*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, pp. 79–84.

[78] Paulius Micikevicius, *3d finite difference computation on gpus using cuda*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, pp. 79–84.

[79] G.E. Moore, *Cramming more components onto integrated circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82–85.

[80] Ajay Narayanan, *Fast binary dilation/erosion algorithm using kernel subdivision*, Proceedings of the 7th Asian conference on Computer Vision - Volume Part II (Berlin, Heidelberg), ACCV'06, Springer-Verlag, 2006, pp. 335–342.

[81] Nikos Nikopoulos and Ioannis Pitas, *An efficient algorithm for 3d binary morphological transformations with 3d structuring elements of arbitrary size and shape*, In Proceedings of 1997 IEEE Workshop on Nonlinear Signal and Image Processing (NSIP'97, 1997.

[82] NVidia, *Nvidia fermi architecture whitepaper.*

[83] ———, *Nvidia geforce 8800 gpu architectural overview.*

[84] ———, *Nvidia geforce gtx 200 gpu architectural overview.*

[85] ———, *Tuning cuda applications for fermi.*

[86] ———, *Nvidia cuda c programming guide 4.0*, 2012.

[87] Matthieu Ospici, Dimitri Komatitsch, Jean-François Mehaut, and Thierry Deutsch, *SGPU 2 : a runtime system for using of large applications on clusters of hybrid nodes*, Second Workshop on Hybrid Multi-core Computing, held in conjunction with HiPC 2011 (Bangalore, India), dec 2011.

[88] NVidia Paulius Mickevicius, *Identifying performance limiters.*

[89] ———, *Multi-gpu programming for finite difference codes on regular grids.*

[90] J Pecht, *Speeding-up successive minkowski operations with bit-plane computers*, Pattern Recogn. Lett. **3** (1985), no. 2, 113–117.

[91] A. Pedron, L. Lacassagne, F. Bimbard, and S. Le Berre, *Parallelization of an ultrasound reconstruction algorithm for non destructive testing on multicore cpu and gpu*, Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, nov. 2011, pp. 1 –8.

[92] Jean philippe Farrugia, Patrick Horain, Erwan Guehenneux, and Yannick Alusse, *Gpucv : A framework for image processing acceleration with graphics processors*, 2012 IEEE International Conference on Multimedia and Expo **0** (2006), 585–588.

[93] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta, *Hierarchical task-based programming with starss*, IJHPCA **23** (2009), no. 3, 284–299.

[94] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta, *A dependency-aware task-based programming environment for multi-core architectures*, Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan, IEEE, 2008, pp. 142–151.

[95] Anis Rahman, Dominique Houzet, and Denis Pellerin, *Visual Saliency Model on Multi-GPU*, GPU Computing Gems Emerald Edition, Elsevier, 2011, pp. 451–472 (Anglais).

[96] Anis Rahman, Dominique Houzet, Denis Pellerin, Sophie Marat, and Nathalie Guyader, *Parallel implementation of a spatio-temporal visual saliency model*, Journal of Real-Time Image Processing **6 special issue** (2010), no. 1, 3–14 (Anglais), Département Images et Signal.

[97] James Reinders, *Intel threading building blocks*, first ed., O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[98] M. C. Rinard, D. J. Scales, and M. S. Lam, *Heterogeneous parallel programming in jade*, Proceedings of the 1992 ACM/IEEE conference on Supercomputing (Los Alamitos, CA, USA), Supercomputing '92, IEEE Computer Society Press, 1992, pp. 245–256.

[99] Y. Roodt, W. Visser, and W. Clarke, *Image processing on the GPU : implementing the canny edge detection algorithm*, Symp. Pattern Recognition Association of South Africa, 2007.

[100] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne, and Daniel Etiemble, *Algorithmic skeletons within an embedded domain specific language for the cell processor*, Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '09, IEEE Computer Society, 2009, pp. 67–76.

[101] Luc Salvo, M. DiMichiel, M. Scheel, P. Lhuissier, B. Mireux, and Michel Suéry, *Ultra fast in situ x-ray micro-tomography : Application to solidification of aluminium alloys*, Materials Science Forum (Volumes 706 - 709) **THERMEC 2011** (2012).

[102] Nadathur Satish, Mark Harris, and Michael Garland, *Designing efficient sorting algorithms for manycore gpus*, Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IPDPS '09, IEEE Computer Society, 2009, pp. 1–10.

[103] Alina Sbîrlea, Yi Zou, Zoran Budimlíc, Jason Cong, and Vivek Sarkar, *Mapping a data-flow programming model onto heterogeneous platforms*, Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (New York, NY, USA), LCTES '12, ACM, 2012, pp. 61–70.

[104] Maraike Schellmann, Jürgen Vörding, Sergei Gorlatch, and Dominik Meiländer, *Cost-effective medical image reconstruction : from clusters to graphics processing units*, Proceedings of the 5th conference on Computing frontiers (New York, NY, USA), CF '08, ACM, 2008, pp. 283–292.

[105] Jean Serra, *Image analysis and mathematical morphology*, Academic Press, Inc., Orlando, FL, USA, 1982.

[106] Ramtin Shams, Parastoo Sadeghi, Rodney Kennedy, and Richard Hartley, *Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images*, Comput. Methods Prog. Biomed. **99** (2010), no. 2, 133–146.

[107] Deshanand Singh, *Implementing fpga design with the opencl standard*, Tech. report, Altera Corporation, November 2011.

[108] Mikhail Smelyanskiy, David Holmes, Jatin Chhugani, Alan Larson, Douglas M. Carmean, Dennis Hanson, Pradeep Dubey, Kurt Augustine, Daehyun Kim, Alan Kyker, Victor W. Lee, Anthony D. Nguyen, Larry Seiler, and Richard Robb, *Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures*, IEEE Transactions on Visualization and Computer Graphics **15** (2009), no. 6, 1563–1570.

[109] Pierre Soille, *On the morphological processing of objects with varying local contrast*, Discrete Geometry for Computer Imagery (Ingela Nyström, Gabriella Sanniti di Baja, and Stina Svensson, eds.), Lecture Notes in Computer Science, vol. 2886, Springer Berlin / Heidelberg, 2003, 10.1007/978-3-540-39966-7_4, pp. 52–61.

[110] Pierre Soille, Edmond J. Breen, and Ronald Jones, *Recursive implementation of erosions and dilations along discrete lines at arbitrary angles*, IEEE Trans. Pattern Anal. Mach. Intell. **18** (1996), no. 5, 562–567.

[111] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton, *Accelerating advanced mri reconstructions on gpus*, Proceedings

of the 5th conference on Computing frontiers (New York, NY, USA), CF '08, ACM, 2008, pp. 261–272.

[112] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu, *Languages and compilers for parallel computing*, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 16–30.

[113] Magnus Strengert, Martin Kraus, and Thomas Ertl, *Pyramid methods in gpu-based image processing*, Workshop on Vision, Modelling, and Visualization VMV '06, 2006, pp. 169–176.

[114] Herb Sutter, *The free lunch is over*, Dr. Dobb's Journal **30** (2005), no. 3.

[115] Alptekin Temizel, Tugba Halici, Berker Logoglu, Tugba Taskaya Temizel, Fatih Omruuzun, and Ersin Karaman, *Gpu computing gems*, Morgan Kaufmann, London, 2011, pp. 547–567.

[116] Julio Toss and Thierry Gautier, *A new programming paradigm for gpgpu*, Euro-Par 2012 Parallel Processing (Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, eds.), Lecture Notes in Computer Science, vol. 7484, Springer Berlin Heidelberg, 2012, pp. 895–907.

[117] Nihan Tuncer, Gürsoy Arslan, Eric Maire, and Luc Salvo, *Investigation of spacer size effect on architecture and mechanical properties of porous titanium*, Materials Science and Engineering : A **530** (2011), no. 0, 633–642.

[118] Leslie G. Valiant, *A bridging model for parallel computation*, Commun. ACM **33** (1990), no. 8, 103–111.

[119] Marc Van Droogenbroeck and Hugues Talbot, *Fast computation of morphological operations with arbitrary structuring elements*, Pattern Recogn. Lett. **17** (1996), no. 14, 1451–1460.

[120] Marcel van Herk, *A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels*, Pattern Recogn. Lett. **13** (1992), no. 7, 517–521.

[121] Luc Vincent, *Morphological transformations of binary images with arbitrary structuring elements*, Signal Process. **22** (1991), no. 1, 3–23.

[122] ———, *Fast grayscale granulometry algorithms*, Mathematical Morphology and its Applications to Image Processing (Fontainebleau, France), EURASIP Workshop ISMM 1994, Kluwer Academic Publishers, 1994, pp. 265–272.

[123] Luc M. Vincent, *Fast opening functions and morphological granulometries*, (1994), 253–267.

[124] V. Volkov, *Better performance at lower occupancy*, GPU Technology Conference 2010, 2010.

[125] ———, *Better performance at lower occupancy*, 2010.

[126] Vasily Volkov and James W. Demmel, *Benchmarking gpus to tune dense linear algebra*, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008.

[127] Ondřej Šťava, Bedřich Beneš, Matthew Brisbin, and Jaroslav Křivánek, *Interactive terrain modeling using hydraulic erosion*, Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (Aire-la-Ville, Switzerland, Switzerland), SCA '08, Eurographics Association, 2008, pp. 201–210.

[128] Lukasz Wesolowski, *An application programming interface for general purpose graphics processing units in an asynchronous runtime system*, Master's thesis, Dept. of Computer Science, University of Illinois, 2008, http ://-charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[129] Michael Wolfe, *Implementing the pgi accelerator model*, Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010 (David R. Kaeli and Miriam Leeser, eds.), ACM International Conference Proceeding Series, vol. 425, ACM, 2010, pp. 43–50.