UN ENVIRONNEMENT PARALLÈLE DE DÉVELOPPEMENT HAUT NIVEAU POUR LES ACCÉLÉRATEURS GRAPHIQUES : MISE EN OEUVRE À L'AIDE D'OPENMP

Gabriel Noaje

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE
ÉCOLE DOCTORALE SCIENCES TECHNOLOGIES ET SANTÉ

THÈSE

présentée par

GABRIEL NOAJE

pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne
Spécialité : Informatique

UN ENVIRONNEMENT PARALLÈLE DE DÉVELOPPEMENT HAUT NIVEAU POUR LES ACCÉLÉRATEURS GRAPHIQUES : MISE EN ŒUVRE À L'AIDE D'OPENMP

soutenue publiquement le 7 mars 2013 devant le jury constitué de :

Directeur :		
	Michaël Krajecki	Professeur, Université de Reims Champagne Ardenne
Co-directeur :		
	Christophe Jaillet	Maître de Conférences, Université de Reims Champagne Ardenne
Rapporteurs :		
	Hervé GUYENNET	Professeur, Université de Franche-Comté
	Zineb HABBAS	Professeur, Université de Lorraine
Examinateurs :		
	Guillaume COLIN-DE-VERDIERE	Chargé mission HPC, CEA
	Olivier FLAUZAC	Professeur, Université de Reims Champagne Ardenne
	Florin POP	Maître de Conférences, Université Politehnica Bucarest
	Adrian TATE	Ingénieur principal, Cray Inc.

Centre de Recherche en STIC

Remerciements

Je tiens à exprimer mes remerciements et toute ma gratitude à Michaël KRAJECKI, Professeur à l'Université de Reims Champagne Ardenne et à Christophe JAILLET, Maître de Conférences à l'Université de Reims Champagne Ardenne, pour avoir accepté d'encadrer cette thèse, pour leur disponibilité et leurs compétences scientifiques que j'ai pu apprécier tout au long de ces années. Ce travail n'aurait jamais pu aboutir sans eux qui ont toujours su me consacrer des moments de leurs temp et de me donner des nombreux conseils.

Je remercie aussi mes rapporteurs Hervé GUYENNET, Professeur à l'Université de Franche-Comté et Zineb HABBAS, Professeur à l'Université de Lorraine, pour avoir rapporté mes travaux, ainsi que les autres membres du jury : Guillaume COLIN-DE-VERDIERE, Chargé mission HPC au Commissariat à l'Energie Atomique, Olivier FLAUZAC, Professeur à l'Université de Reims Champagne Ardenne, Florin POP, Maître de conférences à l'Université Politehnica Bucarest et Adrian TATE, Ingénieur principal chez Cray.

Je remercie mon collègue de bureau Hervé DELEAU pour les discussions techniques qu'on a pu avoir pendant les différents moments de mon travail.

Je tiens également à remercier les ingénieurs du Centre de Calcul Champagne-Ardenne ROMEO, Arnaud RENARD et Hervé DELEAU, pour leur soutien technique et logistique.

Je tiens à remercier tous les membres du département Mathématiques, Mécanique et Informatique, enseignants, ensemble du personnel et doctorants, avec qui j'ai passé des bons moments.

Mes derniers remerciements vont à ma famille, mes parents et spécialement à Cornelia, car sans leur constant soutien et leurs encouragements, je n'aurais pas pu mener à bien cette thèse.

Table des Matières

Table des figures

Liste des tableaux

"Anyone can build a fast CPU.
The trick is to build a fast system."
Seymour Cray

Introduction

L'ÉVOLUTION de la performance des supercalculateurs durant les dernières décennies a été remarquable. La première machine considérée comme un supercalculateur, l'IBM Naval Ordnance Research Calculator, a été utilisée de 1954 à 1963 pour calculer les trajectoires de missiles ; il était capable de réaliser 15 000 opérations par seconde. Un demi-siècle plus tard le Titan du Laboratoire National Oak Ridge dispose de 299 008 cœurs de calcul CPU et 261 632 cœurs de calcul GPU, repartis en 18 688 nœuds multiGPU, et il est capable d'effectuer 17,59 millions de milliards d'opérations par seconde. Il est de nouveau utilisé pour des applications directement liées à la défense (simulation d'armes nucléaires), mais également dans domaine de la recherche : énergie, changement climatique, étude du génome humain ou astronomie. La course n'est pas finie car la demande pour disposer de plus de puissance est toujours plus importante, et même proportionnelle à l'offre. Selon les projections des experts, le domaine du HPC devrait voir des machines franchir la barre de l'exaflopique (1 milliard de milliards d'opérations par seconde) à l'horizon 2018.

Dans la marche inexorable pour des machines de plus en plus performantes, les innovations technologiques dont bénéficient les supercalculateurs sont ensuite rendues accessibles au grand public. Cela fut le cas par exemple des processeurs multi-cœurs qui au début étaient réservés aux supercalculateurs et qui sont aujourd'hui banalisés, au point qu'il est presque impossible d'acheter un ordinateur fixe ou portable, ou un téléphone, dont le processeur ne dispose pas d'au moins deux cœurs de calcul.
Pendant de nombreuses années l'évolution de la puissance des supercalculateurs est venue de celle des processeurs par l'augmentation de leur intégration et de leur vitesse. Cette évolution est désormais limitée par une barrière de consommation énergétique et de dissipation thermique. C'est pour cette raison que le nombre d'unité de calcul des supercalculateurs a été constamment augmenté, pour arriver aujourd'hui à des machines massivement parallèles. Mais cette approche a généré d'autres problèmes, liés notamment aux communications (surcoût des communications par rapport aux calculs), aux réseaux d'interconnexion et à la tolérance aux pannes des processeurs eux-mêmes.

Dans les dernières années le calcul haute performance a adopté une nouvelle tendance

qui promet de rapprocher la barre de l'exascale, par l'utilisation du calcul hybride : les calculs sont accélérés par des accélérateurs matériels, spécialisés et surpuissants, qui sont adjoints aux processeurs dans des nœuds de calcul plus évolués. Les accélérateurs sont des unités massivement parallèles, pouvant être dotées de centaines d'unités de calcul spécialisés, qui se déclinent en plusieurs familles dont les cartes graphiques et les processeurs spécialisés. Les cartes graphiques ont désormais une place très privilégiée dans le paysage HPC grâce à leurs performances crêtes annoncées, qui sont actuellement de l'ordre de plusieurs TFlops pour une seule carte, pour un prix modique.
Les cartes graphiques ont fait leur apparition en même temps que la demande pour des interfaces graphiques faciles à utiliser par l'utilisateur lambda. L'apparition des premiers jeux vidéo a poussé encore plus l'évolution de ces co-processeurs car ils réclamaient de plus en plus de réalisme et de fluidité dans les affichages. À un certain moment ces unités sont devenues si puissantes que des chercheurs ont considéré leur potentiel de puissance, pour les mettre au profit de calculs généralistes indépendants de toute vocation graphique. Des résultats encourageants ont été présentés dès 1990 [LRDG90], mais la conclusion unanime était qu'il y a une barrière importante dans l'adoption de ces accélérateurs à cause de la difficulté de programmation, car tous les calculs devaient être mappés dans le langage de programmation graphique. En 2007, NVIDIA a proposé le premier langage généraliste de programmation pour ses cartes graphiques ; ce fut le point du départ pour une adoption de ces nouvelles architectures dans le monde des supercalculateurs, pour arriver en 2012 à avoir 62 supercalculateurs hybrides dans le TOP500 (12,4%).

L'apport du calcul hybride est significatif et permet en effet d'augmenter considérablement la puissance des machines parallèles. Ainsi l'objectif de l'exascale est effectivement à portée, dans un avenir proche. Toutefois l'augmentation de puissance des supercalculateurs induit une complexification de leur architecture, qui rend toujours plus complexe leur maîtrise. En particulier la programmation des machines hybrides pose de nouveaux challenges. Les différents niveaux de hiérarchie des unités de calcul et de la mémoire rendent ces architectures difficiles à programmer. Une approche multi-niveaux s'impose pour gérer à la fois la communication entre les différents nœuds, entre les différents processeurs et accélérateurs au sein d'un nœud de calcul, et à l'intérieur de chaque accélérateur. Dans le cadre de cette thèse nous nous sommes intéressés aux deux derniers niveaux de cette hiérarchie, et nous proposons des perspectives pour intégrer également la dimension multi-nœuds.

Depuis le début de cette thèse, notre objectif a été de permettre un bon usage des cartes graphiques pour des besoins de calcul haute performance, en suivant donc deux objectifs complémentaires :

– Le premier axe de nos recherches concerne la transformation automatique de code, permettant de partir de code de haut niveau afin de le transformer en un code de bas niveau, équivalent, pouvant être exécuté sur des accélérateurs. Nous avons privilégié le langage de haut niveau OpenMP, le plus répandu pour le parallélisme en mémoire partagée, qui permet d'exprimer le parallélisme en ajoutant des directives dans le code d'origine ; nous ciblons les architectures des GPU NVIDIA, qui sont les plus uti-

lisées dans les supercalculateurs ; le code de bas niveau, en langage CUDA, doit être suffisamment lisible pour qu'il soit possible de le retravailler pour des optimisations ultérieures.
– Par ailleurs, pour donner la possibilité aux utilisateurs d'architectures multiGPU de les utiliser dans de bonnes conditions, il est nécessaire de mettre en place des schémas d'exécution appropriés. Ainsi il faut gérer la communication au sein d'un nœud entre plusieurs cartes graphiques et le CPU d'une manière efficace. Nous avons mener un étude comparative en utilisant le modèle OpenMP et MPI pour piloter les cartes graphiques.

Nous organisons la suite de ce mémoire de la façon suivante : les chapitres trois et quatre se concentrent sur nos contributions dans le cadre de cette thèse, alors que les chapitres un et deux, présentant l'état de l'art, s'avèrent indispensables pour une bonne compréhension des choix réalisés et pour définir le cadre de nos travaux.

Le premier chapitre présente un état de l'art du domaine du calcul à haute performance, pour bien cadrer nos travaux dans le contexte présent. Tout d'abord nous présentons des taxonomies classiques (Flynn, Duncan, Young) concernant les modèles d'exécution des machines parallèles pour avoir une image d'ensemble de leur évolution. Pour continuer nous nous intéressons à la référence actuelle en terme de matériel pour le calcul haute performances, le fameux classement TOP500, dont nous suivons l'évolution et au sein duquel nous voyons l'apparition des architectures hybrides comme un moyen d'atteindre l'exascale. L'intérêt croissant pour les accélérateurs nous amène à présenter en détails les principaux types de matériels existants : GPU (NVIDIA et AMD), co-processeurs spécifiques (ClearSpeed, CellBE, FPGA), ainsi que le dernier Intel Xeon Phi. Nous verrons qu'une connaissance fine de l'architecture des accélérateurs s'avère indispensable pour les exploiter.

Le deuxième chapitre fait la transition dans le niveau logiciel, notamment les langages de programmation pour les accélérateurs, en liaison étroite avec les architectures matérielles présentées auparavant. Nous commençons avec une présentation générale des langages de programmation parallèle, pour passer ensuite aux langages de programmation spécialisés (d'abord de bas niveau, ensuite de haut niveau). Nous constaterons que chaque catégorie a ses avantages et ses inconvénients. Nous concluons ce chapitre par une présentation des transformateurs de code, pour bien cadrer le premier axe de recherche de nos contributions.

Le troisième chapitre détaille notre outil de transformation de code. Nous avons choisi d'utiliser OpenMP comme langage de haut niveau et nous transformons le code dans le langage CUDA de NVIDIA. Les lignes directrices des transformations seront axées sur ces deux langages, mais sans perdre de généralité car les étapes utilisées se prêtent aussi bien pour générer du code OpenCL, par exemple. Le chapitre débute avec une présentation de l'outil OMPi qui se trouve à la base de notre transformateur de code. Aussitôt après nous introduisons les spécificités du langage CUDA et leur intégration

dans l'outil. Une importante partie du chapitre est alors consacrée aux approches de transformation des boucles parallèles : ce sont en effet les éléments qui dégagent le plus de parallélisme et qui sont donc de bons candidats pour être accélérés sur des GPU. Une des contraintes que nous imposons à notre outil est de générer un code suffisamment lisible par un humain pour qu'il puisse ensuite être retravaillé. Néanmoins certaines améliorations du code transformé peuvent être réalisées automatiquement dans le cadre de l'outil et nous dédierons une section pour les présenter et pour donner des pistes pour leur implémentation.

Dans le quatrième et dernier chapitre nous présentons une étude comparative pour la mise en place d'un environnement capable de prendre en charge des architectures multi-accélérateurs au sein d'un même nœud. La gestion des contextes et des communications dans le cadre d'un environnement multiGPU pose des problèmes et doit être soigneusement réalisée. Nous présentons les différents possibilités et les comparons, puis nous donnons les détails d'implémentation nécessaires à une implémentation efficace ; la mise en œuvre sur un exemple classique d'application met en évidence l'avantage d'utiliser la programmation en mémoire partagée avec OpenMP. L'approche fait partie d'un modèle multi-niveaux capable de gérer une grappe de multiGPU, pour tirer profit des futures architectures exascale.

Enfin nous concluons sur l'ensemble de ce travail et nous présentons des perspectives à court, moyen et plus long terme, en vue d'être prêts pour l'avènement des machines hybrides à l'époque de l'exascale.

I

Vers des machines exaflopiques

Sommaire

Il y a quelques dizaines d'années les traitements parallèles constituaient un domaine exotique réservé à des chercheurs. Cette perception a changé peu à peu et a été récemment balayée avec la démocratisation des processeurs multi-cœurs, disponibles dans presque tous les ordinateurs grand public actuels ou encore dans les dispositifs nomades de type téléphone intelligent (*smartphone*). Les processeurs quadri-cœurs sont devenus la norme aujourd'hui et cela augmente la pression sur les développeurs, qui doivent « réfléchir parallèle » pour proposer de nouvelles fonctionnalités et tirer profit de la performance disponible.
La seule augmentation de la vitesse des unités de calcul amenant à une barrière énergétique, la suite logique était de multiplier les éléments de calcul pour ainsi obtenir des supercalculateurs, machines gigantesques pouvant contenir jusqu'à des centaines de milliers de processeurs fonctionnant de concert [1]. Dans ce chapitre nous présenterons les différents modèles d'exécution qui se sont dégagés au fil du temps, et nous donnerons un aperçu des supercalculateurs les plus puissants du moment.

Dans ce paysage de machines surpuissantes il y a de nouveaux arrivés, qui ont trouvé une place à part : les accélérateurs, notamment les cartes graphiques. Avec une évolution fulgurante nourrie par les exigences des *gamers* pour un rendu toujours plus réaliste, les cartes graphiques sont bientôt devenues des architectures d'une puissance considérable, capables d'exécuter des dizaines de milliards d'opérations par seconde. Les chercheurs se sont intéressés à ces accélérateurs *dormant* dans les machines des utilisateurs, pour les détourner de leur but initial et les faire travailler sur des calculs généralistes. Le point d'inflexion s'est produit en 2007 suite au lancement du langage de programmation CUDA pour les cartes graphiques NVIDIA. A partir de ce moment les architectures graphiques ont été rendues plus facilement accessibles pour des calculs généralistes, ce qui a posé les bases du *GPU Computing*. Nous consacrons la seconde partie de ce chapitre aux particularités des architectures des principales familles d'accélérateurs matériels (dont les cartes graphiques), car une connaissance fine des spécificités des architectures cibles est indispensable pour toute démarche de programmation et en particulier pour fournir des codes capables d'exploiter la puissance de calcul disponible.

I.1 Modèles d'exécution

Lorsqu'on considère le domaine de la programmation séquentielle, le modèle d'exécution des programmes est celui de la machine, théorique, de von Neumann. Il permet de concevoir des programmes indépendants de la machine cible. On peut énumérer de nombreux modèles de programmation : algorithmique classique, programmation logique, programmation fonctionnelle, programmation orientée objets, ... Sur une machine sé-

1. La barrière énergétique constatée est liée à la fois aux difficultés de refroidissement des processeurs, qui chauffent d'autant plus que la fréquence d'horloge augmente, et à la consommation absolue des systèmes, qui s'avèrera finalement critique et favorisera l'avènement des accélérateurs matériels pour seconder les processeurs dans des nœuds de calcul hybrides au sein des supercalculateurs.

quentielle, il y a donc une distinction très forte entre modèle d'exécution (unique) et modèle de programmation. Cette distinction n'existe pas dans le domaine du parallélisme, qui compte par ailleurs une grande variété de supports possibles. On se base sur des modèles pour en extraire les caractéristiques essentielles, en ce qui concerne les composants physiques et les mécanismes logiques de conception et d'analyse des applications. Pour l'étude du parallélisme, on distingue généralement trois niveaux d'abstraction :

– Les modèles d'architecture sont des modèles de bas niveau, introduits pour simplifier la vue des mécanismes physiques, en identifiant les points clés du comportement du système.
– Les modèles d'exécution, de niveau intermédiaire, ont pour rôle d'étudier les mécanismes liés à la gestion des ressources. Ils sont liés au modèle d'architecture sous-jacent.
– Au plus haut niveau se trouvent les modèles de programmation, qui permettent d'exprimer le parallélisme dans les applications, en liaison avec les modèles de niveaux inférieurs.

Ces modèles sont intimement liés et la frontière qui les sépare n'est pas fixée nettement. Nous les exposerons ici en considérant que les modèles d'exécution caractérisent les types d'architectures sur lesquels ils sont utilisables.

I.1.1 Taxonomie de Flynn

Sur une machine séquentielle, l'unique modèle d'exécution est celui de la machine de von Neumann : le flot d'instructions (unique) est exécuté au rythme d'une instruction par cycle d'horloge, chaque instruction intervenant sur une donnée unique (scalaire). Dans le domaine du calcul parallèle, une application peut correspondre à plusieurs flots d'instructions différents, sur un unique ensemble de données ou sur des flots de données différents. La classification proposée par Michael J. Flynn [Fly66, Fly72] se base sur ces distinctions ; elle met en évidence quatre modèles d'exécution, comme le montre la figure I.1.

1) SISD : *Single Instruction stream - Single Data stream*

Ce modèle d'exécution est celui du monde séquentiel pour des ordinateurs à processeur scalaire. Un programme est un flot d'instructions sur un flot de données, comme décrit précédemment (voir aussi la figure I.2). Les machines mono-processeur pipelinées ou super-scalaires entrent également dans cette catégorie, en relevant plus du traitement simultané que de l'exécution concurrente [FR96].

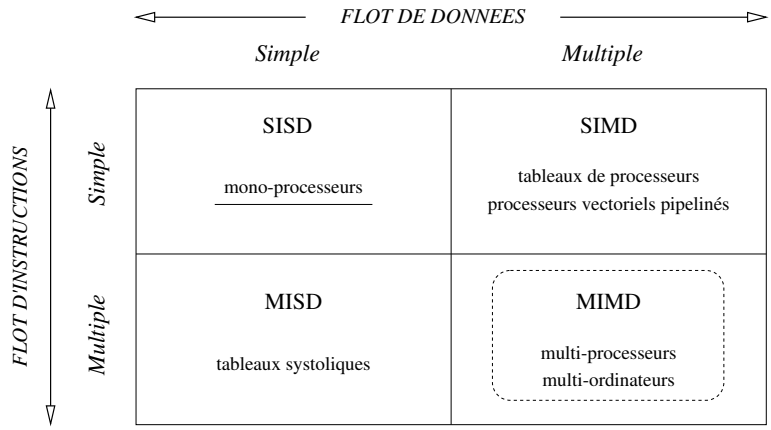
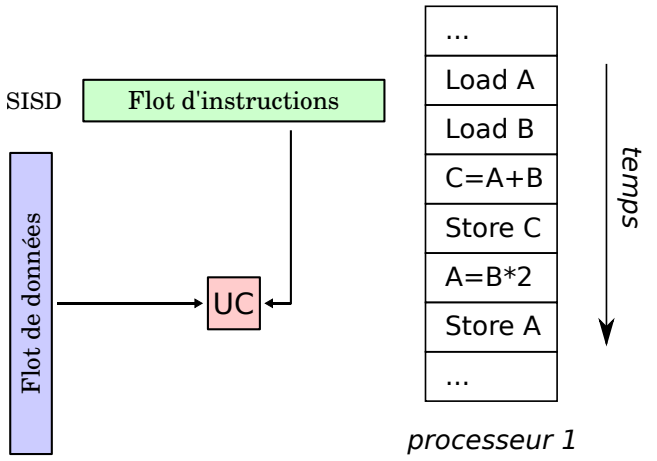


FIGURE I.1: Taxonomie de Flynn

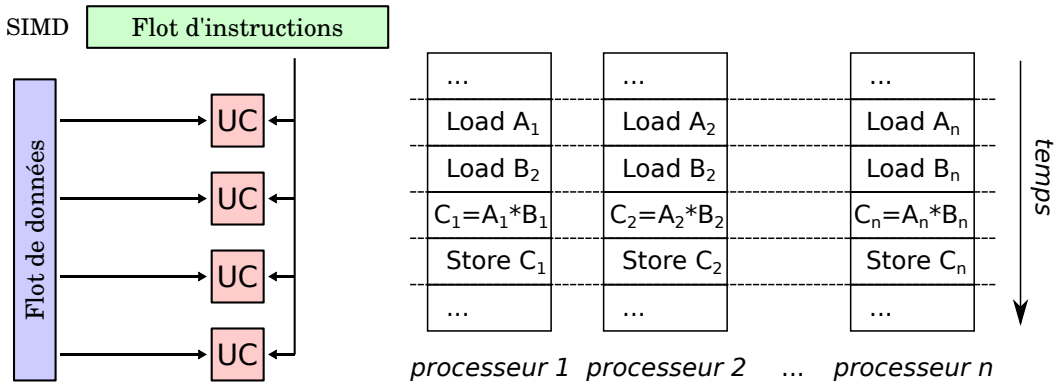


(a) Détails de l'architecture (b) Gestion de flots d'instructions

FIGURE I.2: Taxonomie de Flynn : modèle SISD

2) SIMD : *Single Instruction stream - Multiple Data stream*

Dans ce modèle d'exécution, le même flot d'instructions est exécuté de façon concurrente par différents processeurs, chacun sur un flot de données (figures I.1 et I.3). Les instructions s'exécutent de façon synchrone.
C'est en dérivant ce modèle qu'on décrit le mieux l'exécution multi-threadée en vigueur sur les cartes graphiques : le modèle SIMT sera présenté en détail dans les parties suivantes (voir I.3.1, 1)).

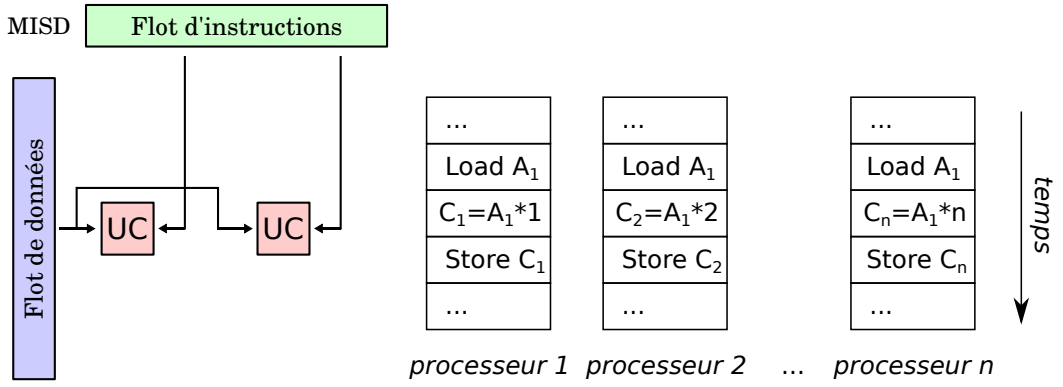


(a) Détails de l'architecture (b) Gestion de flots d'instructions

FIGURE I.3: Taxonomie de Flynn : modèle SIMD

3) MISD : *Multiple Instruction stream - Single Data stream*

Tel qu'il est décrit dans [FR96], le modèle d'exécution MISD correspond à des architectures composées de plusieurs unités fonctionnelles, qui travaillent chacune leur tour sur un même jeu de données, de façon synchrone. C'est typiquement le cas des tableaux systoliques de processeurs (chaque processeur correspond à une unité fonctionnelle) ; on peut se représenter le transfert des données comme si elles étaient "aspirées" d'une unité fonctionnelle à l'autre (l'image est celle du cœur, qui pompe le sang) (figure I.4).



(a) Détails de l'architecture (b) Gestion de flots d'instructions

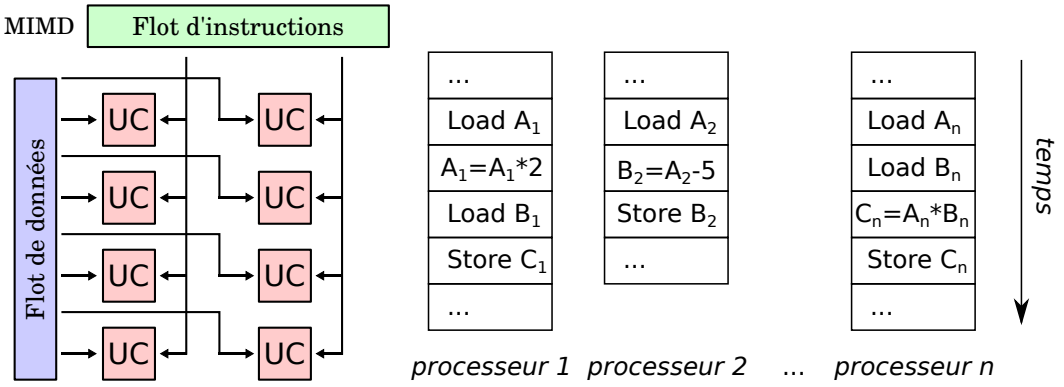
FIGURE I.4: Taxonomie de Flynn : modèle MISD

Un tel modèle d'exécution est adapté à des applications très spécifiques qui peuvent être issues de différents domaines, par exemple :

- dans le domaine du traitement d'image, il permet d'exécuter plusieurs traitements consécutifs sur une image ;
- des machines à trier ont été conçues sur ce modèle : les unités fonctionnelles, rudimentaires, absorbent des valeurs qu'elles transmettent à l'un de leurs voisins selon leur ordre ; on implémente également ainsi des files de priorités : à chaque étape, la valeur entrée transite vers sa position et la plus petite valeur est extraite [Qui03].

4) MIMD : *Multiple Instruction stream - Multiple Data stream*

Il s'agit du modèle le plus large pour exprimer le parallélisme : plusieurs processeurs sont nécessaires, et chacun exécute son propre flot d'instructions, de façon autonome et sur un flot de données qui lui est propre (figure I.5). Ce modèle offre le plus de possibilités, en permettant de concevoir des applications constituées de tâches, dépendantes ou non, exécutables en parallèles ; mais les communications et/ou synchronisations entre les processeurs doivent être gérées explicitement, et il faut équilibrer la charge répartie sur les processeurs.



(a) Détails de l'architecture (b) Gestion de flots d'instructions

FIGURE I.5: Taxonomie de Flynn : modèle MIMD

I.1.2 Modèles d'architecture relevant de MIMD - classifications de Duncan et Young

Duncan [Dun90] a proposé des modifications à la taxonomie de Flynn pour appréhender les nouveaux types des architectures selon les différentes méthodes d'accès physique à la mémoire. Selon sa classification il y a deux classes génériques (figure I.6) :
– architectures à mémoire partagée ;
– architectures à mémoire distribuée.
Les modèles à mémoire partagée donnent une vue de la mémoire selon un unique espace d'adressage : il s'agit de la vue qu'en ont les utilisateurs, et les applications qu'ils développent. Quant aux modèles à mémoire distribuée, ils ne prévoient pas que la mémoire puisse être utilisée de façon partagée : dans ce modèle, chaque processeur dispose nécessairement de sa mémoire localement. En bas niveau, les échanges entre les processeurs (communication de données, synchronisation) doivent alors forcément s'effectuer par échange de messages.
Le tableau I.1 présente d'une manière synthétique les avantages et les inconvénients de ces deux types d'architecture.

En partant du type d'interconnexion, Young et al. [YTR+87] proposent une classification plus fine des architectures selon l'accès physique à la mémoire :

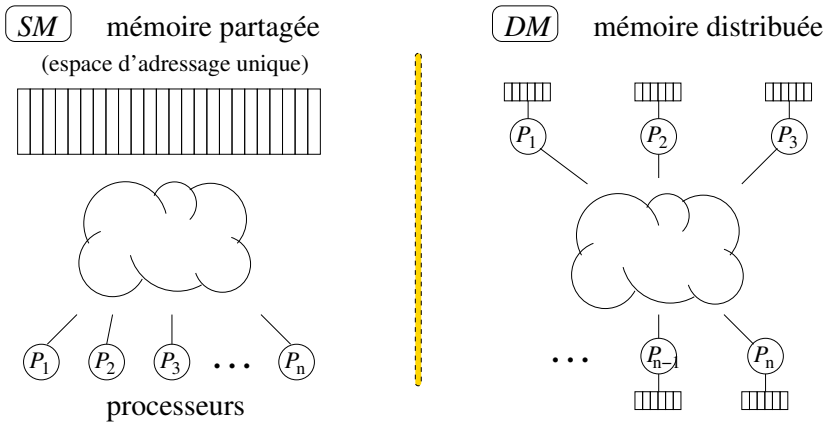


FIGURE I.6: Architecture à mémoire partagée *vs.* mémoire distribuée

	Mémoire partagée	Mémoire distribuée
Avantages	– l'espace d'adressage global permet un accès mémoire facile *via* un modèle de programmation intuitif – le partage des données se fait de manière rapide et uniforme grâce au rapprochement de la mémoire par rapport au processeur	– passage à l'échelle facile avec l'augmentation du nombre de processeurs – la taille mémoire augmente proportionnellement avec le nombre des processeurs – chaque processeur a accès à sa propre mémoire sans coût supplémentaire pour assurer la cohérence des caches
Inconvénients	– passage à l'échelle difficile car l'ajout des processeurs peut augmenter géométriquement le trafic sur les interconnexions mémoire-processeurs – responsabilité de l'utilisateur pour faire la synchronisation – coût considérable pour la conception de machine avec un nombre conséquent des processeurs	– difficile à programmer : l'utilisateur doit se charger des toutes les communications inter-processeurs – temps d'accès non-uniforme – difficile de remodeler la répartition de données

TABLE I.1: Avantages et inconvénients des architectures à mémoire partagée et distribuée

– *architectures UMA (Uniform Memory Access)* : la mémoire est entièrement partagée et tous les processeurs y accèdent en temps constant (et très court) ;
– *architectures NUMA (Non-Uniform Memory Access)* : la mémoire est logiquement partagée mais séparée et placée en différents endroits et avec des temps d'accès inégaux [2] ;
– *architectures NoRMA (No Remote Memory Access)* : chaque processeur dispose d'une mémoire locale (à laquelle il accède sans aucune pénalité), mais il n'y a aucune possibilité d'accéder à la mémoire des autres processeurs.

On peut enfin présenter la classification proposée par Eric E. Johnson [Joh88], dictée par les différentes méthodes d'accès physique à la mémoire ainsi que par les mécanismes de communication/synchronisation. Elle donne une approche théorique encore plus générale pour les architectures relevant du modèle d'exécution MIMD, et prévoit globalement quatre catégories d'architectures :
– *Global Memory-Shared Variables* (GMSV) - machines à mémoire partagée ;
– *Distributed Memory-Shared Variables* (DMSV) - machines hybrides ;
– *Distributed Memory-Message Passing* (DMMP) - machines à passage de messages ;
– *Global Memory-Message Passing* (GMMP).
Ces quatre catégories sont toutes théoriquement envisageables mais
– les architectures GMMP cumulent les inconvénients liés à leur passage à l'échelle et les difficultés de programmabilité ; en pratique il n'y a aucun intérêt à concevoir de telles machines, qui n'existent que dans la théorie
– à l'inverse les machines DMSV cumulent les avantages ; leur conception physique est celle d'une architecture à mémoire distribuée, à laquelle est adjoint un système permettant de considérer la mémoire comme partagée (DSM, *Distributed Shared Memory*) : la mémoire partagée distribuée peut être implémentée au niveau matériel (H-DSM, *Hardware*-DSM) ou *via* une surcouche logicielle (S-DSM, *Software*-DSM ; V-DSM, *Virtually*-DSM). L'inconvénient de telles architectures est que leur conception ne passe pas à l'échelle : en effet le système de DSM offre des temps d'accès à la mémoire extrêmement dégradés lorsque la taille de la machine augmente.

I.2 Évolution des machines parallèles

I.2.1 Évolution des processeurs

1) Loi de Moore

Le premier microprocesseur (Intel 4004) a été inventé en 1971. Il s'agissait d'une unité de calcul de 4 bits, cadencée à 108 kHz et intégrant 2300 transistors. Depuis, la capa-

2. Lorsque la cohérence et la consistance des accès mémoire est réalisée par un système de caches, on parle de machines CC-NUMA (*Cache-Coherent Non-Uniform Memory Access*).

cité d'intégration des transistors et la diminution de la gravure n'ont pas cessé d'être améliorés chaque année, ce qui a permis d'augmenter les performances des processeurs.

L'observation de cette évolution constante à permis à Gordon Moore de constater en 1965 que la complexité des semiconducteurs proposés en entrée de gamme doublait tous les ans à coût constant depuis 1959, date de leur invention. Il postulait la poursuite de cette croissance au moins pour les 10 années à venir. Cette augmentation exponentielle fut rapidement nommée Loi de Moore [Moo65].

Dans les années suivantes Moore a altéré sa prédiction, notamment en 1975, pour prédire une doublement tous les *deux* ans [Moo75].

Actuellement la forme la plus connue de la loi du Moore est énoncée ainsi :
Le nombre de transisteurs dans les circuits intégrés doublera environ tous les deux ans.

Bien qu'il ne s'agisse pas d'une loi physique mais juste d'une extrapolation empirique, cette prédiction s'est révélée étonnamment exacte. Entre 1971 et 2001, la densité des transistors a doublé chaque 1,96 année (figure I.7).

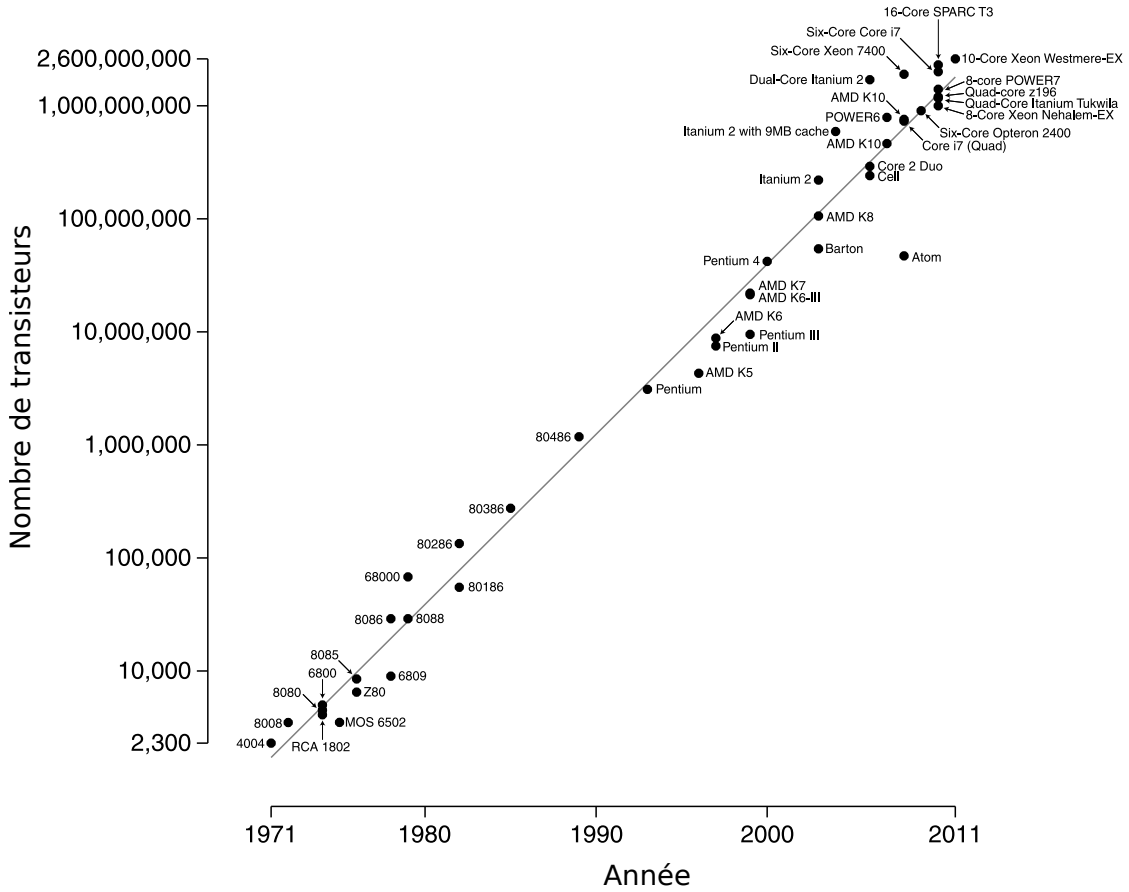


Figure I.7: Loi du Moore (© Wgsimon)

2) Processeurs multi-cœurs

Une autre interprétation (erronée) de la loi de Moore annonce un doublement de la fréquence d'horloge tous les 18 mois. Cela a été à peu près vérifié depuis 1973, et aurait dû théoriquement continuer encore jusqu'en 2015, avant d'atteindre des limites physiques. En effet, depuis 2004, la fréquence des processeurs tend à stagner en raison de difficultés de dissipation thermique, qui empêchent une montée en fréquence en dépit de la taille plus faible des composants.

Désormais, pour gérer au mieux la puissance dissipée et contourner cette limite les constructeurs de processeurs ont cessé d'augmenter la fréquence d'horloge pour en revanche augmenter le nombre de cœurs des processeurs. Le principe est présenté par la figure I.8 :

- Considérons une augmentation de 20% de la fréquence d'horloge pour un processeur : nous pouvons constater une augmentation de 13% de sa performance mais de 73% pour sa consommation.
- Si pour le même processeur la fréquence baisse de 20%, sa performance baisse de 13% et sa consommation de 49%.
- Ainsi deux processeurs avec une fréquence moindre consomment autant qu'un processeur avec une fréquence normale, mais ont une puissance cumulée bien plus importante : pour une même consommation électrique (+ 2%) nous pouvons donc utiliser deux cœurs au sein d'un unique processeur, qui vont offrir une performance globale augmentée de 73% par rapport au processeur mono-cœur.

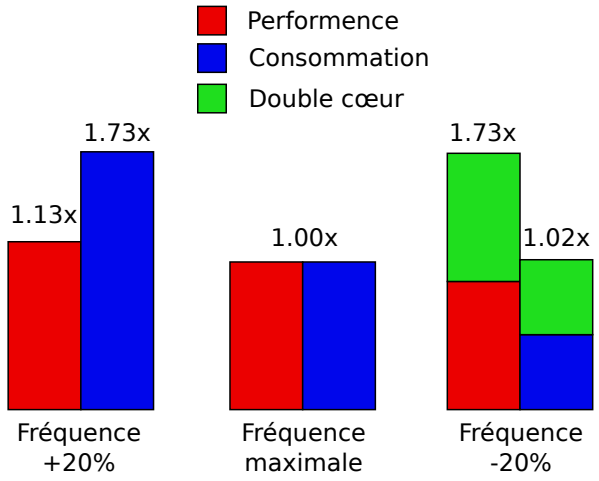


FIGURE I.8: Comparatif des performances entre un processeur mono cœurs et un processeur multi-cœurs

3) Loi d'Amdahl

La multiplication de cœurs a commencé en 2005 avec des processeurs dual-cœurs et elle a continué depuis, jusqu'à des processeurs octo-cœurs. Cependant l'augmentation en

performance crête est juste théorique car cette multiplication a un impact extrêmement important sur le développement logiciel qui doit être spécialement adapté pour tirer parti du matériel.
Le gain possible est toujours limité par le rapport du code qui peut être exécuté en parallèle pour tirer profit de tous les cœurs disponibles. Ce phénomène est connu sous le nom de loi d'Amdahl.

> Considérons f la portion du code qui peut être parallélisée et $(1-f)$ la portion séquentielle : l'accélération maximale qui peut être atteinte en utilisant p processeurs est $Speedup(f,p) = \frac{1}{(1-f)+\frac{f}{p}}$, comme illustré par la figure I.9.

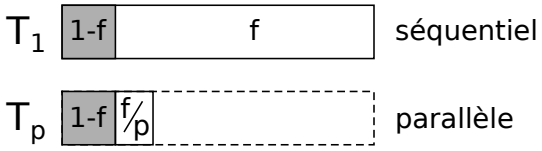


Figure I.9: Loi d'Amdahl (représentation schématique)

Il y a aussi deux corollaires naturels à la loi d'Amdahl :
– quand f est proche de zéro, l'optimisation n'aura presque aucun effet ;
– quand p augmente à l'infini, l'accélération est limitée par $1/(1-f)$.
Ainsi, bien qu'il s'agisse d'un lieu commun, il faut préciser que l'utilisation de processeurs multi-cœurs ne peut être avantageuse pour une application qu'à condition de commencer à y mettre en évidence tout le parallélisme possible, bien avant de chercher à en optimiser le code.

La multiplication des cœurs à l'intérieur du processeur impose de relever des défis non seulement au niveau de modèle de programmation, mais aussi au niveau de l'architecture, car les problèmes a résoudre sont nombreux : mettre en œuvre un réseau d'interconnexion rapide et fiable entre les cœurs, assurer la cohérence des différents niveaux de mémoire, *etc.*
Ainsi, en continuant de s'appuyer sur des cœurs de calcul évolués, les processeurs multi-cœurs se limitent actuellement à l'utilisation de 8 cœurs de calcul (processeurs Intel Xeon E7-2820, AMD FX-8150 par exemple).

4) Accélérateurs many-cœurs

Les accélérateurs ont suivi un développement similaire, en utilisant des cœurs de calcul avec une architecture réduite. Ils peuvent intégrer des centaines ou des milliers de cœurs, pour former des architectures dites many-cœurs. Soumis à la même limitation par la loi d'Amdahl, les accélérateurs sont conçus pour exécuter des algorithmes hautement parallèles (*embarassingly parallel*) et donc avec peu de communications sur un très grand nombre de données.

Dans la partie I.3 : *Accélérateurs*, nous ferons une présentation approfondie des différentes catégories d'accélérateurs many-cœurs, en donnant à la fois des détails structurels

et concernant leurs modèles d'exécution spécifiques (exécution et synchronisation des threads).

Des architectures hybrides mettent en commun, au sein d'un même nœud de calcul, des processeurs multi-cœurs et des accélérateurs many-cœurs. La mutualisation de ces ressources a pur but d'augmenter la capacité de calcul tout en réduisant la consommation énergétique, mais ça n'est pas sans poser des problèmes car beaucoup de nouvelles questions apparaissent, sur lesquelles nous reviendrons par la suite (chapitre II : *Approches de programmation des accélérateurs*).

I.2.2 TOP500

Augmenter la puissance d'un seul nœud de calcul est vite devenu insuffisant, car les perspectives de puissance ont rapidement suscité l'intérêt pour des problèmes de plus en plus "gros", et pour de nouvelles catégories de problèmes. La suite logique était donc de mettre en commun des nœuds de calcul et de les faire communiquer pour résoudre des grands challenges (dynamique des fluides, raisonnement et intelligence artificielle, *etc*). Initialement cette mise en commun s'est faite sur la base de machines grand publique, pour réaliser des clusters généralistes (*beowolf*) allant jusqu'à quelques centaines de nœuds. Aujourd'hui des supercalculateurs de dizaines de milliers de nœuds sont utilisés, dans les domaine académique, militaire ou commercial, pour accélérer des calculs qu'il serait impossible de résoudre autrement.

Le TOP500 est un projet de classification des 500 supercalculateurs les plus puissants au monde. Il est à l'initiative de chercheurs européens et américains : Hans Meuer de l'Université de Mannheim en Allemagne, Jack Dongarra de l'Université du Tennessee à Knoxville, Erich Strohmaier et Horst Simon du National Energy Research Scientific Computing Center (NERSC) du Lawrence Berkeley National Laboratory (LBL).
Depuis juin 1993, le projet met à jour sa liste tous les six mois :
- en juin lors de la conférence ISC (*International Supercomputing Conference*) ;
- en novembre à l'occasion de la conférence *SuperComputing* (actuellement *International Conference for High Performance Computing, Networking, Storage and Analysis*).

1) Métrique

LINPACK désigne à la fois une bibliothèque de programmation pour l'algèbre linéaire (maintenant supplantée par LAPACK) et un test de performance permettant d'évaluer la performance des ordinateurs pour les calculs en virgule flottante [Don79]. Pour la mise en place du test, Jack Dongarra est parti du principe que les ordinateurs doivent être évalués sur une application représentative du calcul scientifique. Il propose donc un programme de calcul matriciel, consistant à résoudre un système linéaire dense $n \times n$: le système $Ax = b$ compte n équations à n inconnues, et est résolu par la méthode du pivot de Gauss en $\frac{2}{3}n^3 + n^2$ opérations à virgule flottante ; le temps mesuré pour le

calcul permet de déduire un nombre d'opérations par seconde, qui s'exprime en FLOPS (GFlops, TFlops, ...) C'est sur la base de ce test, Linpack Benchmark, que sont établis les classements du TOP500.

Il s'agit bien sûr d'une simplification, puisqu'aucun numéro unique ne peut refléter la performance globale d'un système informatique [3]. Néanmoins, la performance au Linpack permet de fournir une bonne correction de la performance crête officielle fournie par les fabricants [4] [DLP03].

On se doit d'évoquer au moins deux alternatives au Linpack, qui permettent d'établir des classements selon un autre point de vue :

– La plus avancée est l'initiative du comité Graph500, qui propose un nouveau benchmark basé sur des problèmes issus de la théorie des graphes [MWBA10]. Plus de 50 personnes, des milieux industriel et académique, se sont réunies pour proposer des tests synthétiques dans des domaines variés : sécurité des systèmes d'information, informatique médicale, réseaux sociaux et réseaux symboliques. La première version du top Graph500 a été présentée lors de SC2010. La dernière version, du juin 2012, met en première position le supercalculateur Mira de l'ANL (Argonne National Laboratory, Etat-Unis), alors que le Top500 le classe en troisième position.
– La puissance augmentant, la consommation électrique des machines parallèles prend des proportions importantes et peut devenir un frein à l'accroissement de puissance. C'est pourquoi le coût d'exploitation d'un système tient compte du rapport de sa puissance et de sa consommation. De ce fait une nouvelle liste, nommée Green500, à été introduite en novembre 2007 : elle propose une classification des supercalculateurs par leur efficacité énergétique, selon la métrique *FLOPS/Watt* [FC07, FS09]. Nous verrons que les accélérateurs, ayant une performance crête intéressante pour une faible consommation énergétique, permettent aux supercalculateurs hybrides d'être bien positionnés dans le classemen Green500. On note par exemple que le supercalculateur d'Intel équipé de cartes MIC qui se retrouve seulement à la 150ème position dans le TOP500 du juin 2012, est classé 21ème du classement Green500.

Actuellement les trois *tops* (TOP500, Green500 et Graph500) sont dominés par les machines BlueGene/Q, qui offrent manifestement la meilleure performance en termes de puissance crête et d'efficacité énergétique.

3. A sa sortie le test du Linpack a été critiqué principalement pour le fait qu'il ne teste que la résolution de systèmes denses linéaires, ce qui n'est pas absolument représentatif de toutes les applications de calcul intensif sensibles en calcul scientifique, et car il fournit des niveaux de performance généralement impossibles à obtenir sauf à mettre en œuvre toutes les optimisations envisageables pour l'application en vue d'une exécution sur cette machine cible spécifiquement.

4. La performance crête théorique est obtenue sur la base de la fréquence de la machine, en cycles par seconde, multipliée par le nombre d'opérations qu'elle peut réaliser en un cycle ; la performance réelle sera toujours inférieure à la performance crête théorique.

2) Classement et analyse

Pour les machines parallèles classées au sein du TOP500, on dispose de différentes données complétant le rang obtenu pour la machine considérée, dont notamment :
- le nombre de cœurs de calcul dont dispose la machine, #Processors ;
- la performance théorique en GFlops de la machine, R_{peak} ;
- les valeurs pratiques obtenues par le test : R_{max}, la performance Linpack maximale mésurée(en GFlops) pour le plus gros problème tournant sur la machine ; N_{max}, la taille du problème considéré.

Le tableau I.2 présente la tête du dernier classement TOP500, en date de novembre 2012.

En regardant en arrière, depuis 1993, on peut observer que la performance du supercalculateur figurant à la première position a doublé tous les 14 mois, ce qui est en accord avec la loi du Moore (figure I.10). Cette évolution est généralement dûe aux avancées technologiques dans la production des processeurs. Néanmoins, dans les dernières années, de plus en plus des systèmes hybrides (avec des accélérateurs) ont fait leur apparition en haut du classement. En particulier la machine qui détenait la première place du TOP500 de novembre 2012 était une machine hybride, composée de 18 688 cartes graphiques NVIDIA Tesla K20.
En novembre 2012, le TOP500 recense 12,4% de machines hybrides (Tableau I.3). Parmi ceux-ci 50 supercalculateurs sont basés sur des cartes graphiques NVIDIA, alors qu'il n'y avait qu'un seul en juin 2010. En comparant les segments 101-500 des classements TOP500 de novembre 2011 et novembre 2012 nous pouvons même constater une explosion de 700% des machines disposant d'accélérateurs NVIDIA, ce qui montre à quel point cette architecture se démocratise auprès des chercheurs.

I.2.3 Perspective / prospective

L'évolution récente du calcul parallèle a été fortement liée à l'arrivée de processeurs multicœurs dans les supercalculateurs, puis à leur utilisation systématique. Ces processeurs se sont généralisés à tel point que toutes les machines actuelles du TOP500 les utilisent, mais également presque tous les ordinateurs personnels actuels, portables ou fixes.
Pour l'utilisation des cartes graphiques le processus fut exactement inverse. Après une démocratisation des GPU dans presque tous les ordinateurs personnels grâce à l'industrie des jeux vidéo, les chercheurs ont commencé à s'intéresser à leur puissance et à la possibilité de les utiliser pour réaliser des calculs généralistes. Comme nous l'avons vu précédemment les accélérateurs, et les cartes graphiques en particulier, ont fait leur entrée dans le TOP500 depuis quelques années. De plus en plus de supercalculateurs utiliseront bientôt des accélérateurs, et les utilisateurs se conformeront aux modèles de programmation correspondants.

L'avenir proche des supercalculateurs se trouve sans doute lié à plusieurs pistes, indépendantes, dans des domaines différents. Nous pouvons sans doute en énumérer

Rang	**Nom**	**Etablissement**	**Fabriquant**	**Nb de cœurs**		**Performance** (GFlops)		**Conso.**
				Total	Accél.	R_{peak}	R_{max}	(kW)
1	Titan	DOE/SC/Oak Ridge National Laboratory	Cray Inc.	560640	261632	27112550	17590000	8209
2	Sequoia	DOE/NNSA/LLNL	IBM	1572864	0	20132659.2	16324751	7890
3	K computer	RIKEN Advanced Institute for Computational Science (AICS)	Fujitsu	705024	0	11280384	10510000	12660
4	Mira	DOE/SC/Argonne National Laboratory	IBM	786432	0	10066330	8162376	3945
5	JUQUEEN	Forschungszentrum Juelich (FZJ)	IBM	393216	0	5033165	4141180	1970
6	SuperMUC	Leibniz Rechenzentrum	IBM	147456	0	3185050	2897000	3423
7	Stampede	Texas Advanced Computing Center/Univ. of Texas	Dell	204900	112500	3958965	2660290	-
8	Tianhe-1A	National Supercomputing Center in Tianjin	NUDT	186368	100352	4701000	2566000	4040
9	Fermi	CINECA	IBM	163840	0	2097152	1725492	822
10	DARPA Trial Subset	IBM Development Engineering	IBM	63360	0	1944391.68	1515000	3576

TABLE I.2: Les 10 supercalculateurs les plus puissants au monde (TOP500, novembre 2012)

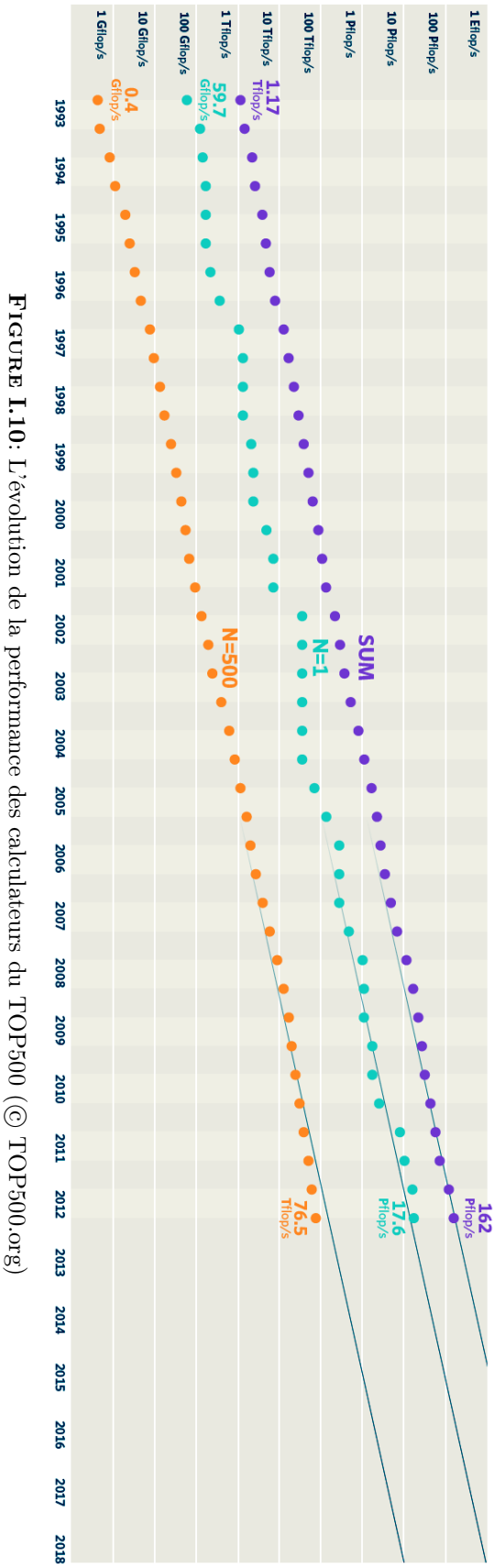


FIGURE I.10: L'évolution de la performance des calculateurs du TOP500 (© TOP500.org)

Accélérateurs	Nombre	%	Coeurs	R_{peak} (GFlops)	R_{max} (GFlops)
Aucun	438	87,6	124844008	165167770	12795391
GPU NVIDIA	50	10,0	30996814	54428191	1553476
Intel Xeon Phi	7	1,4	4302764	6309356	337301
GPU AMD	3	0,6	827300	1832963	62832
IBM Cell	2	0,4	1168500	1537632	136800

TABLE I.3: Machines hybrides dans le TOP500 (novembre 2012)

quelques unes.

– Le processeur Intel Xeon Phi, dernier né de chez Intel, s'annonce comme un concurrent redoutable pour les cartes graphiques, pas encore nécessairement grâce à sa puissance crête, mais plutôt pour sa facilité de programmation. En effet, alors qu'il est difficile de programmer pour tirer profit de la puissance crête théorique des différents accélérateurs actuels (conçus par essence pour des traitements non généralistes), les nouveaux accélérateurs d'Intel se présentent comme des processeurs pouvant être utilisés comme des accélérateurs, et sont conçus pour être utilisés comme des processeurs classiques du point de vue de la programmation : ils ne nécessitent pas l'apprentissage d'un langage de programmation spécifique, et peuvent être utilisés de façon similaire si on veut les considérer comme des processeurs ou comme des accélérateurs.
– Dans le même esprit, une autre piste envisagée réside dans l'unification des accélérateurs avec les processeurs pour obtenir des unités de calcul accélérées (APU - *Accelerated Processing Unit*). C'est d'ailleurs le modèle proposé par les processeurs AMD Fusion.
– Le futur du calcul parallèle sera non seulement une course pour obtenir plus de puissance crête, mais aussi pour relever le défi de l'efficacité énergétique, car le coût des projets des supercalculateurs ne doit pas basculer de l'investissement (acquisition) vers le fonctionnement (entretien, maintenance, alimentation énergétique). C'est pour cette raison qu'il y a un intérêt croissant pour des processeurs basse consommation, comme les processeurs Atom (Intel), Tegra (NVIDIA) ou ARM (*Advanced Risc Machine*, distribués par ARM Ltd).

C'est sans doute du passage à l'échelle de ces nouvelles générations d'accélérateurs et/ou processeurs, et de leur agrégation au sein de machines parallèles, que naîtront les architectures au sommet des différents *tops* de demain. Mais avant d'envisager le passage à l'échelle il est nécessaire d'avoir une connaissance approfondie de l'architecture des accélérateurs actuels, pour pouvoir en tirer un maximum de performance. La section suivante présentera en détail les familles d'accélérateurs représentatives, tant du point de vue matériel ainsi que du point de vue du modèle d'exécution associé.

I.3 Accélérateurs

Jusqu'à présent nous avons présenté les modèles d'exécution en vigueur sur les machines à architectures séquentielle et parallèle, considérées comme caractérisées par la structuration de l'ensemble de leurs processeurs (CPU) et leur fonctionnement. Cependant, comme nous l'avons évoqué, les processeurs classiques (monocœurs ou multicœurs) sont à la limite de leur évolution et le potentiel de croissance pour la puissance des machines parallèles réside actuellement dans l'adjonction d'accélérateurs aux processeurs des nœuds de calcul. Nous allons donc maintenant présenter les architectures des accélérateurs qui prévalent actuellement, en commençant par les GPU, qui ont ouvert la voie avec l'arrivée de processeurs graphiques grand public il y a bientôt trente ans.

I.3.1 GPU (NVIDIA, ATI)

Au début des années 1980 le calculateur Apple Lisa était le premier à proposer une interface graphique. À partir de ce moment la popularité grandissante des systèmes d'exploitation à caractère graphique a créé un marché pour un nouveau type de processeurs. En 1985, le Commodore Amiga était le premier ordinateur personnel livré avec un processeur graphique. Le périphérique avait son propre jeu d'instructions pour dessiner des lignes et remplir des polygones, et incluait un *stream processor* pour accélérer le mouvement et la manipulation des images bitmap. Dès lors les accélérateurs graphiques 2D, intégrant des jeux d'instructions spécifiques afin d'améliorer l'affichage et la fluidité des systèmes d'exploitation graphiques, commencèrent à se démocratiser.

À peu près à la même époque, dans le milieu professionnel, la société SGI (*Silicon Graphics, Inc*) popularisa l'utilisation des graphismes en trois dimensions dans un grand nombre de domaines - applications gouvernementales, militaires, scientifiques, médicales, *etc.* SGI était le leader dans le domaine des stations de travail 3D industrielles et, grâce à son idée de rendre son interface de programmation libre de droit et indépendante des plateformes, le passage de ces technologies vers l'informatique personnelle n'était plus alors qu'une question de temps.

Au milieu des années 1990, la demande de graphismes en 3D en informatique personnelle s'est rapidement accrue. L'apparition de jeux de tir subjectif comme *Doom*, *Wolfenstein 3D* et *Quake* imposa de créer progressivement des environnements 3D plus réalistes pour les jeux. À la même époque, des sociétés comme NVIDIA, ATI Technologies et 3dfx Interactive ont commencé à produire des accélérateurs graphiques à des prix suffisamment abordables pour intéresser le grand public. Les évolutions ultérieures des accélérateurs graphiques ont toujours été poussées par une demande de plus en plus importante de la part de l'industrie des jeux vidéo. À la même époque Microsoft lançait sa propre interface de programmation 3D, nommée DirectX, dans le cadre de la version 95 de Windows, pour concurrencer OpenGL.

La puissance des accélérateurs graphiques a eu une croissance exponentielle dans les années suivantes. Pour la première fois, les développeurs pouvaient contrôler les trai-

tements effectués sur leurs GPU, car de nouveaux modèles proposaient à la fois des vertex programmables et des nuances de pixels programmable (*shading*). L'apparition de GPU permettant de programmer les traitements déclencha de nombreuses recherches sur la possibilité d'utiliser les circuits graphiques pour autre chose que pour faire du graphisme pur. Ce moment fut celui de la naissance du calcul générique sur un processeur graphique (GPGPU, *General-Purpose Processing on Graphics Processing Unit*). Par contre l'approche générale de la programmation des premiers traitements GPU était extraordinairement compliquée. Les API graphiques standard comme OpenGL et DirectX étant toujours le seul moyen d'interagir avec un GPU, l'exploitation des ressources de calcul nécessitait de repenser les algorithmes dans une perspective graphique, et de transformer les instructions de calcul en des opérations graphiques natives utilisant ces API.

Au début des années 2000, les cartes graphiques étaient conçues pour faire de l'affichage, c'est-à-dire produire une couleur et déterminer la position (x,y) sur l'écran d'un point provenant d'un espace virtuel 3D, à l'aide d'unités arithmétiques programmables appelées *pixel shaders* et *vertex*. Les programmeurs se rendent cependant compte du fait qu'ils peuvent programmer les *shaders* pour qu'ils considèrent les entrées comme des données numériques signifiant autre chose que des couleurs. Les résultats étaient restitués par le GPU comme étant la couleur finale d'un pixel alors que celle-ci était, en fait, le résultat du calcul que le programmeur avait demandé au GPU d'effectuer sur ses entrées. L'idée était de mapper les calculs dans le pipeline graphique, ainsi les données devenaient des textures et des vertex et les résultats, des points, interprétés ensuite comme des résultats numériques. Le GPU fut donc détourné de son contexte d'affichage en lui faisant effectuer des traitements qui n'avaient rien à voir avec du graphisme. Cependant cette approche n'a pas séduit la grande masse des programmeurs, du fait de ses contraintes : nombreuses restrictions pour faire interpréter les données numériques comme des données graphiques ; et obligation, pour espérer tirer profit de la puissance des GPU, d'acquérir de nouvelles connaissances *a priori* indépendantes (graphisme, langage de *shading*, ...).

C'est en 2007 qu'a lieu la réelle révolution du GPGPU, lorsque NVIDIA a mis à disposition des programmeurs la première version de son framework de programmation, appelé CUDA (*Compute Unified Device Architecture*), qui permet un accès direct à l'architecture du périphérique *via* le langage C, familier, auquel sont apportés des modifications minimales. Cela fut le point de départ pour le *GPU Computing* avec, par la suite, de nouvelles versions améliorant la programmabilité [ND10]. Comme nous le verrons également (voir II.2 : *Langages de programmation spécifiques pour accélérateurs graphiques*) d'autres API de programmation sur GPU se développent au même moment : elles constituent des alternatives aux propositions de NVIDIA, rendant accessibles les autres familles d'accélérateurs graphiques : C++ AMP pour AMD ; puis OpenCL, par Kronos, pour améliorer la portabilité en offrant une API ouverte aux architectures hétérogènes CPU-GPU et/ou variées (GPU NVIDIA ou AMD, processeur Cell, ...).
Dès lors les GPU deviennent donc attractifs et on peut espérer tirer parti de ces accélé-

rateurs, intéressants pour leur grande puissance de calcul théorique, leur prix modeste, ainsi que pour leur faible consommation malgré un système de refroidissement efficace.

Cependant les architectures des GPU ne correspondent à aucun des modèles d'exécution traditionnels dérivés de la taxonomie de Flynn ; ainsi la conception des algorithmes parallèles doit-elle être abordée avec beaucoup de soin. Pour une meilleure compréhension de leur fonctionnement et pour souligner les particularités de chacun de ces nouveaux accélérateurs, les sections suivantes présentent en détail les architectures actuelles des constructeurs majeurs dans le domaine. Par la suite nous présenterons des accélérateurs spécifiques, qui constituent des alternatives aux GPU (accélérateur ClearSpeed, CellBE ; FPGA), et nous aborderons les processeurs MIC, qui intègrent une partie des technologies issues du GPU au sein même du CPU.

1) NVIDIA - Kepler

La série Tesla est le haut de gamme de NVIDIA dédié au calcul généraliste [LNOM08]. Cette section présente la dernière architecture de cette série, nommée Kepler. Depuis le lancement de son framework CUDA, Kepler est la troisième architecture de GPU NVIDIA ; elle a été explicitement conçue pour le calcul à haute performance.

ARCHITECTURE MATÉRIELLE

Contenant jusqu'à 7,1 milliards de transistors gravés dans la nouvelle technologie de lithographie à 28 nm, l'architecture Kepler offre une performance en double précision allant jusqu'à 1 TFlops. Pour comparer avec la génération précédente, l'architecture Fermi disposait de 3 milliards de transistors dans une puce gravée avec la technologie 40 nm, pour une performance en double précision moindre de moitié.

Les détails techniques des dernières architectures Fermi et Kepler sont présentés dans le tableau I.4 [NVIe, NVId].

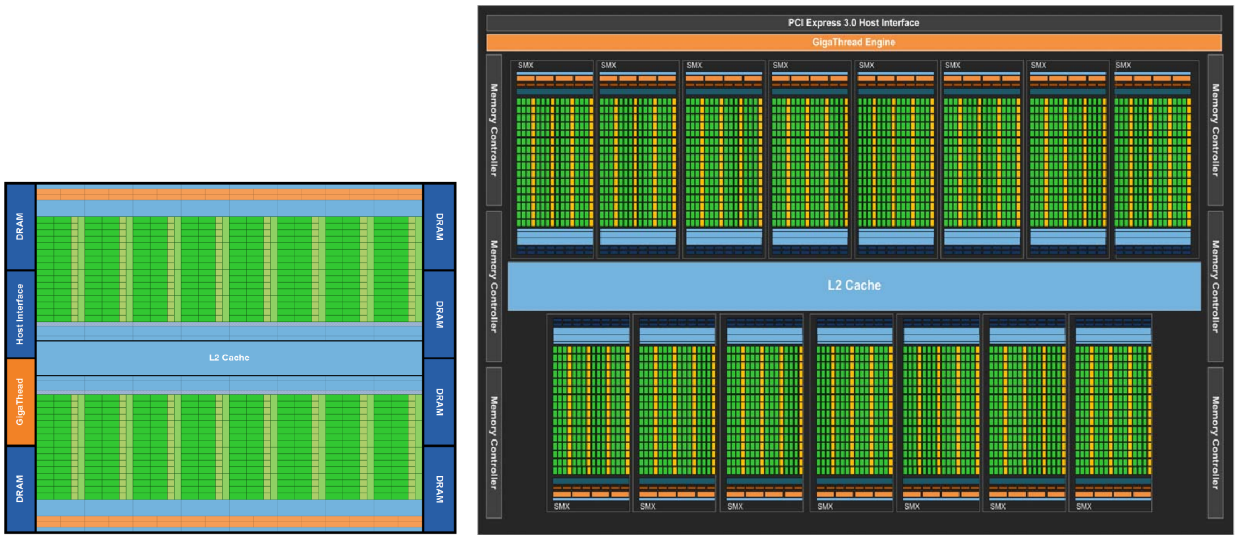
L'architecture Kepler du GK110 est composée de 2880 unités de calcul élémentaires (les *cœurs CUDA*), organisées en 15 *Streaming Multiprocessors* (SMX) de 192 cœurs chacun. En comparaison l'architecture Fermi ne comptait que 512 cœurs, organisées en 16 SM (*Streaming Multiprocessors*, version Fermi) de 32 cœurs chacun (figure I.11a). L'architecture Kepler est actuellement déclinée en deux architectures : le K10 (512 cœurs) composé de 2 cartes GK104 ; K20 (2496 cœurs) et K20X (2688 cœurs) composé d'une carte GK110 (voir figure I.11b).

L'architecture d'un *Streaming Multiprocessor* (SM/SMX) a beaucoup changé au cours du temps. Celle du Kepler (figure I.12b) est composée de 192 cœurs CUDA en simple précision et 64 unités en double précision, 32 unités de SFU (*Special Function Units* [5]), 32 unités d'entrées/sorties (dédiées aux accès mémoire).

5. dédiées aux approximations des fonctions transcendantales `exp`, `log`, `sin` et `cos`.

Caractéristique	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute capability	2.1	3.0	3.5
Nombre de cœurs	512	1536	2880
Fréquence d'horloge (MHz)	650	745	745
Performance (crête) en simple précision (GFlops)	1331	2288	3950
Performance (crête) en double précision (GFlops)	665	95	1310
Taille mémoire (GB)	6	4	8
Bande passante mémoire (GB/sec)	177	160	250

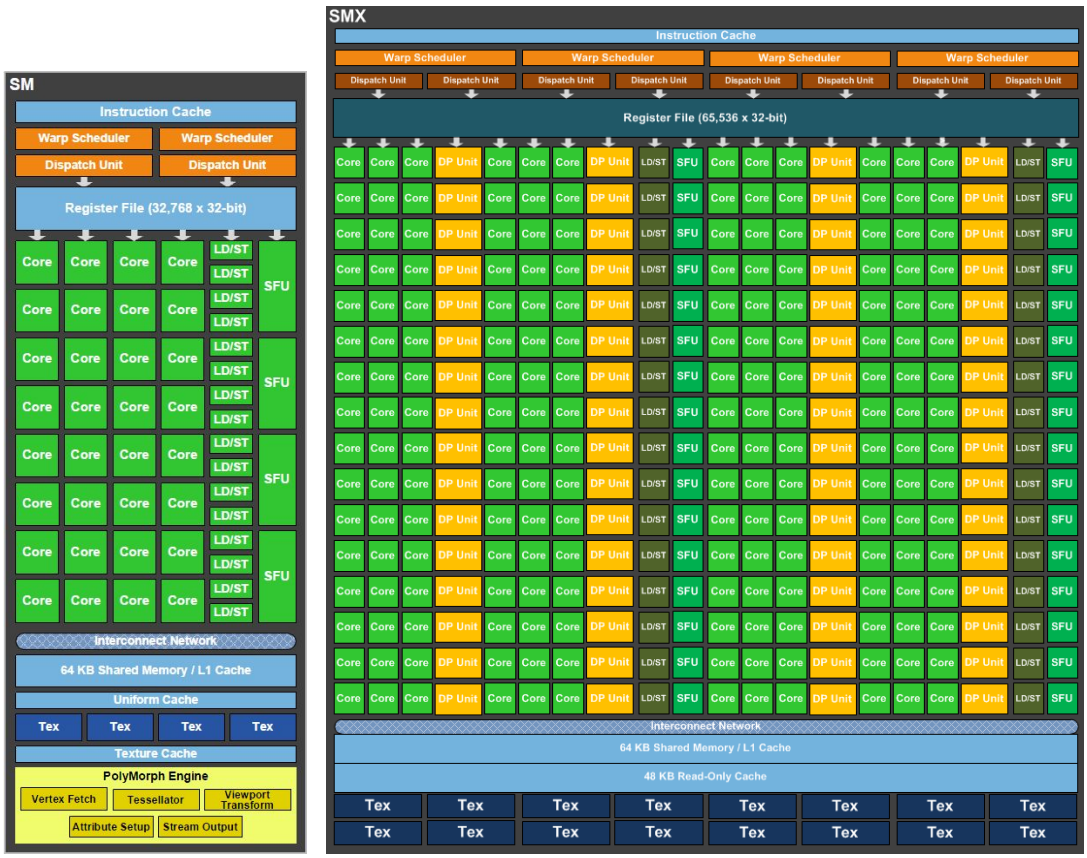
TABLE I.4: Détails techniques de la série Tesla (architectures Fermi et Kepler)



(a) Fermi GF104 (mi 2010) (b) Kepler GK110 (fin 2012)

FIGURE I.11: Fermi *vs.* Kepler : Architecture (© NVIDIA)

Un aspect innovant de l'architecture du Kepler réside dans la réduction de la fréquence d'horloge des shaders. En effet dans l'architecture Fermi le shader avait une fréquence double par rapport à celle du GPU. Cela permettait de soutenir un certain débit avec moins de copies de la part de l'unité d'exécution. Ainsi les ordonnanceurs et les unités de texturing opéraient à la fréquence des processeurs élémentaires (SP, *Streaming Processor*), alors que les unités d'entrées/sorties (pour des lectures/écritures) et les SFU opéraient à la fréquence du shader, donc deux fois plus vite que la fréquence du cœur. Mais cette implémentation est gourmande en énergie. C'est pour cette raison que dans l'architecture Kepler, les shaders fonctionnent à la même fréquence que le reste du cœur. Pour maintenir le même débit, la plupart des unités fonctionnelles ont été



(a) Fermi SM (GF104) (b) Kepler SMX (GK110)

FIGURE I.12: Fermi *vs.* Kepler : architecture d'un *Streaming Multiprocessor* (© NVIDIA)

doublées afin de compenser la baisse de fréquence. Ainsi, les architectures de la gamme Kepler sont bien deux fois plus puissantes que les Fermi, mais pas quatre fois comme on l'annonce parfois.

MODÈLE D'EXÉCUTION

Le modèle d'exécution SIMT (*Single Instruction stream, Multiple Threads*) proposé par NVIDIA est proche du modèle SIMD (*Single Instruction stream, Multiple Data stream*), mais il présente des particularités importantes. L'unité d'exécution du code s'appelle un *kernel* (*noyau* en français [6]). Le kernel est exécuté simultanément sur un ou plusieurs SM ou SMX par l'intermédiaire de threads regroupés dans des blocs (*blocks* : au plus 1024 threads/bloc). Chaque thread dispose de son propre environnement d'exécution (espace d'adresses, registres d'état, mémoire locale). Ils sont cependant regroupés dans des *warps*, lots de 32 threads qui exécutent tous leurs instructions de façon synchrone.

6. Dans toute la littérature scientifique le terme anglais est utilisé sans créer de confusion avec le noyau GNU/Linux ; c'est pour cette raison que nous l'utiliserons tout au long du manuscrit.

Chaque SMX contient 4 ordonnanceurs pour les warps et 8 unités pour l'envoi des instructions (*instruction dispatch units*) permettant à 4 warps de s'exécuter simultanément sur 2 instructions différentes au même cycle. L'ordonnanceur est composé d'unités matérielles spécialisées offrant par exemple :
- un mécanisme de scoreboarding pour les registres dans le cas des opérations avec une latence importante (accès aux textures et chargement mémoire) ;
- un mécanisme de décision pour choisir le meilleur warp parmi une liste des warps éligibles ;
- un mécanisme d'ordonnancement au niveau des blocs.

Caractère innovant de l'architecture du Kepler : En plus de ces structures matérielles, une importante partie de la puce de l'architecture Fermi était dédiée à un mécanisme de scoreboarding pour éviter les aléas d'acheminement des données pour les calculs mathématiques. Selon plusieurs études, les architectes ont conclu que cette information est déterministe (les latences du pipeline mathématique ne sont pas variables) et qu'il serait possible pour le compilateur de déterminer à l'avance l'instant où une instruction est prête à être exécutée, et de stocker cette information dans l'instruction elle-même. Cela a permis de remplacer des unités matérielles complexes par des unités plus simples, capables d'extraire les informations prédéterminées de la latence et de les utiliser pour déterminer les warps éligibles pour être exécutés.

HIÉRARCHIE MÉMOIRE

Pendant l'exécution du kernel, les threads ont accès à différents espaces mémoire :
- *shared memory* : 48Ko de mémoire partagée disponible sur chaque SMX, qui permet de collaborer à l'ensemble des threads qui s'y exécutent. Cette mémoire partagée, organisée en 32 banques (*banks*, indépendantes pour accélérer les accès concurrents) et alimentée avec 16Ko cache L1 [7]. Cette mémoire partagée bénéficie d'une latence réduite ; étant implémentée sur la puce, la vitesse d'accès est équivalente à celle des registres [tant qu'il n'y a pas de conflit de banque entre les threads].
- *local memory* : un maximum de 521Ko de mémoire privée disponible pour chaque thread [8], utilisée pour stocker des tableaux ou structures trop volumineuses pour être mises dans l'espace des registres (*register spill*) ; cette zone mémoire se retrouve dans la mémoire globale du périphérique et est dont pénalisée par des temps de latence très important pour le chargement des données, bien que les dernières architectures favorisent les délais d'accès par l'utilisation de deux niveaux de caches mémoires (L1 et L2).
- *global memory* : un maximum de 6Go pour l'ensemble de la carte GPU, répartis en 8 partitions partagées par tous les threads pendant toute la durée d'exécution de l'application ; l'accès passe par des caches L1 et L2, ainsi que par un cache de

7. ou 16Ko de mémoire partagée par SMX avec 48Ko cache L1, puisqu'une partie (32Ko) est reconfigurable

8. 16Ko pour les ancienne architectures, de *compute capability* inférieure à 2.0

données en lecture seule.
- *constant memory* : 64Ko de mémoire, en lecture seule, situés dans la mémoire globale du périphérique. L'accès est accéléré grâce à 8Ko de cache spécifique par SMX mais, comme pour la mémoire locale, les temps d'accès sont très pénalisants.
- *texture memory* : il s'agit d'un espace mémoire en lecture seule, situé dans la mémoire globale, optimisé pour des accès en deux dimensions (car accessible *via* les unités de texture). Comme la mémoire constante, cette mémoire est située dans la mémoire globale de la carte et les temps d'accès sont donc défavorables, bien que les accès soient accélérés par l'utilisation de caches mémoire spécifiques (huit caches L2 de 32Ko, un par partition DRAM ; 10 caches L1 de 24KB chacun, associés aux TPC, *Texture Processor Cluster*).

Les caractéristiques de ces accélérateurs sont prévues pour les rendre très performants dans le domaine de l'accélération des calculs hors du processeur, et ce pour un ensemble de problèmes de plus en plus large. Cela est favorisé par les possibilités offertes par l'API CUDA, en constante évolution. Cependant leur structuration très forte et les spécificités de leur hiérarchie mémoire demandent encore un investissement fort aux programmeurs qui souhaitent en exploiter toute la puissance potentielle, et les rend donc difficilement programmables pour des utilisateurs non avertis. C'est pour cette raison que nous avons travaillé au développement d'outils permettant de simplifier le travail des programmeurs pour la conception de leurs codes CUDA (voir le chapitre III : *Transformation automatique de code OpenMP en code CUDA*).

2) AMD - Radeon HD

AMD a présenté la nouvelle architecture de ses GPU en décembre 2011. Elle tranche avec les générations précédentes et correspond à la volonté d'AMD d'une remise à plat de l'architecture de ses cartes graphiques, afin de la rendre hiérarchique en vue du passage à l'échelle. Nous présentons ici les principales caractéristiques de l'architecture selon les trois aspects suivants : structuration physique de l'architecture, hiérarchie de threads et hiérarchie mémoire.

Les processeurs graphiques AMD de dernière génération (série Tahiti) sont basés sur l'architecture GCN (AMD *Graphics Cores Next* : figure I.13) [Adv12]. Ils sont gravés en 28 nm, ce qui permet d'intégrer jusqu'à 4,3 milliards de transistors. Synthétiquement, les principales caractéristiques techniques du modèle phare de la série, l'AMD Radeon HD 7970 GHz, sont les suivantes (tableau I.5) :

ARCHITECTURE : STRUCTURATION PHYSIQUE

Nous présentons les éléments de l'architecture en nous basant sur les deux images suivantes : la figure I.13 présente les éléments caractéristiques de l'architecture globale, structurée en *compute units* ; la figure I.14, quant à elle, montre les détails d'un de ces *compute units*.

Caractéristique	Valeur
Nombre de cœurs	2048
Fréquence d'horloge (MHz)	1050
Performance (crête) en simple précision (GFlops)	4300
Performance (crête) en double précision (GFlops)	1075
Taille mémoire (GB)	3
Bande passante mémoire (GB/sec)	288

TABLE I.5: Détails techniques du AMD Radeon HD 7970 GHz

En partant de la structure globale pour aller vers l'intérieur des composants il convient de mentionner schématiquement les éléments suivants, dans l'ordre :
- le Radeon HD 7970 implémente l'architecture GCN ;
- il comprend 32 *Compute Units* (CU), qui sont le lieu des exécutions parallèles ; on notera que les CU sont structurés par groupes de quatre au sein de *clusters* pour mettre en commun une partie des caches de niveau 1 (partie en lecture seule) [9] ;
- un CU, qui correspond à ce que sont les SMX des architectures Kepler (NVIDIA), est composé de 4 unités SIMD (*SIMD units*) et leur permet de partager différentes ressources nécessaires à leur fonctionnement : unité de contrôle de flux d'instructions (branchement, fetch), caches d'instructions et de données (niveau L1), *front-end*
- une unité SIMD dispose de 16 unités vectorielles (*vectorial ALU*, l'équivalent des *CUDA cores* d'une architecture Kepler de NVIDIA) et de ses propres registres pour permettre l'exécution des threads en parallèle : registre d'instruction et compteur ordinal (uniques puisque qu'à un instant donné un seul groupe de threads peut s'exécuter sur l'unité, et qu'ils exécutent leurs instructions de façon synchrone), et registres de calcul.

Au total, l'architecture GCN d'un AMD Radeon HD 7970 dispose donc de 2048 coeurs de calcul élémentaires ($32 \times 4 \times 16$).

EXÉCUTION : HIÉRARCHIE DE THREADS

Sur la base de la structuration physique hiérarchique présentée ci-dessus, on obtient la structuration suivante pour les threads qui s'exécutent sur la carte graphique :
- Les 16 unités vectorielles d'une unité SIMD permettent l'exécution de 16 instructions élémentaires, simultanément mais sur des données différentes, donc le traitement de 64 données pendant les 4 cycles d'horloge nécessaires au traitement complet d'une instruction élémentaire. On parle de *wavefront* (en français « front d'onde », qui serait le pendant d'un *warp* de la terminologie NVIDIA) ; La taille des *wavefronts* est donc de 64 threads.
- Par ailleurs une unité SIMD peut disposer de 10 wavefronts à ordonnancer pour l'exécution (choix du prochain à "passer"), pouvant même éventuellement provenir

9. Les clusters des GCN sont l'équivalent des TPC de l'architecture Kepler de NVIDIA.

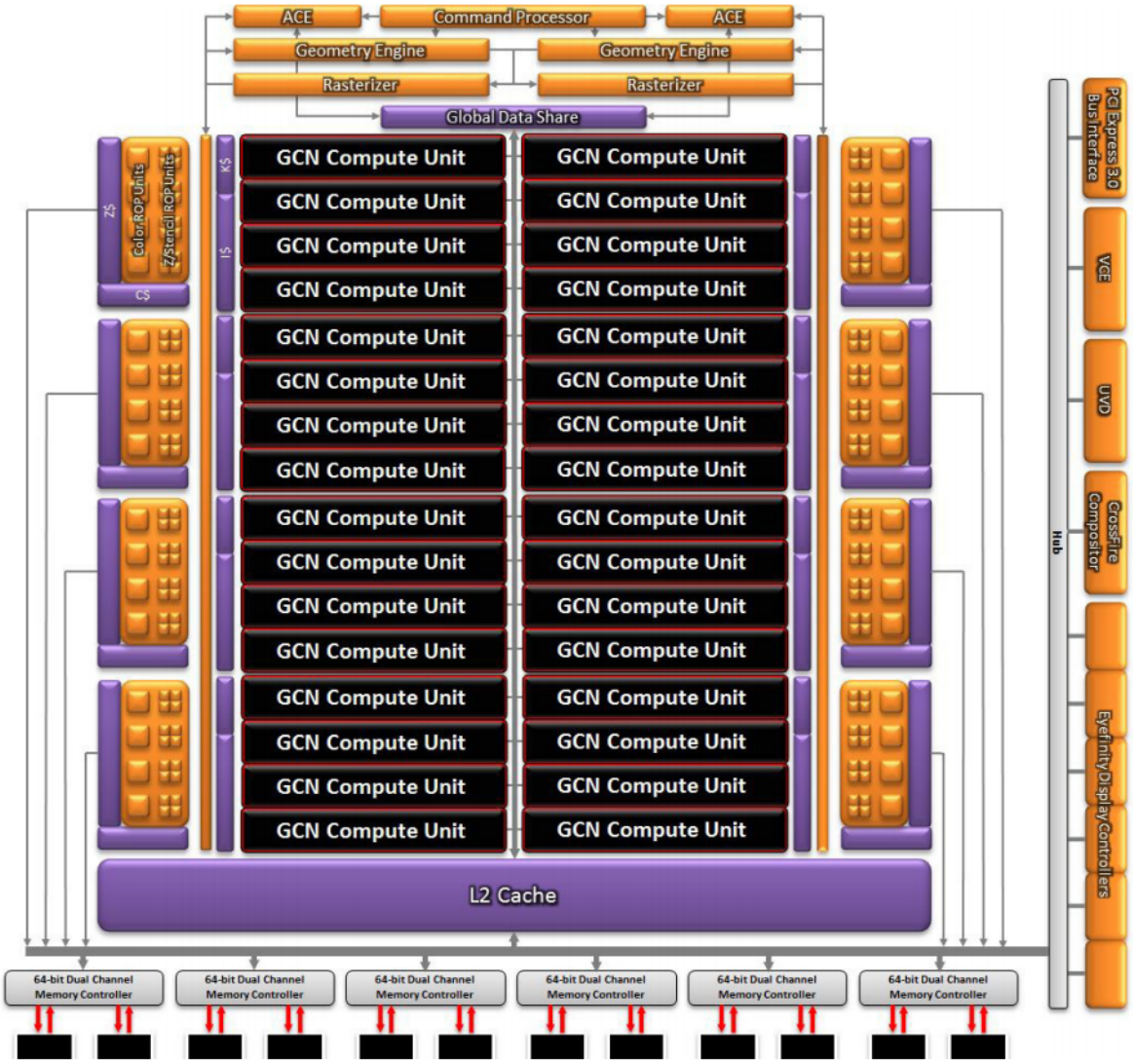


FIGURE I.13: AMD Radeon HD 7970 : Architecture GCN (*Graphics Core Next*) (© AMD)

de kernels différents.
Et comme un CU dispose de 4 unités SIMD on peut disposer simultanément, sur un CU, de 40 wavefronts de 64 threads.

– On notera que, de même que NVIDIA introduit les *blocks* comme un niveau de structuration formel permettant aux programmeurs de considérer des ensembles de threads disponibles pour une exécution sur un SMX, AMD propose la notion de *work-group* rassemblant plusieurs wavefronts.
– Enfin, comme l'architecture GCN d'un AMD Radeon HD 7970 dispose de 32 CU, on peut y avoir jusqu'à 81920 threads différents disponibles sur une carte à un moment donné ($32 \times 40 \times 64$).

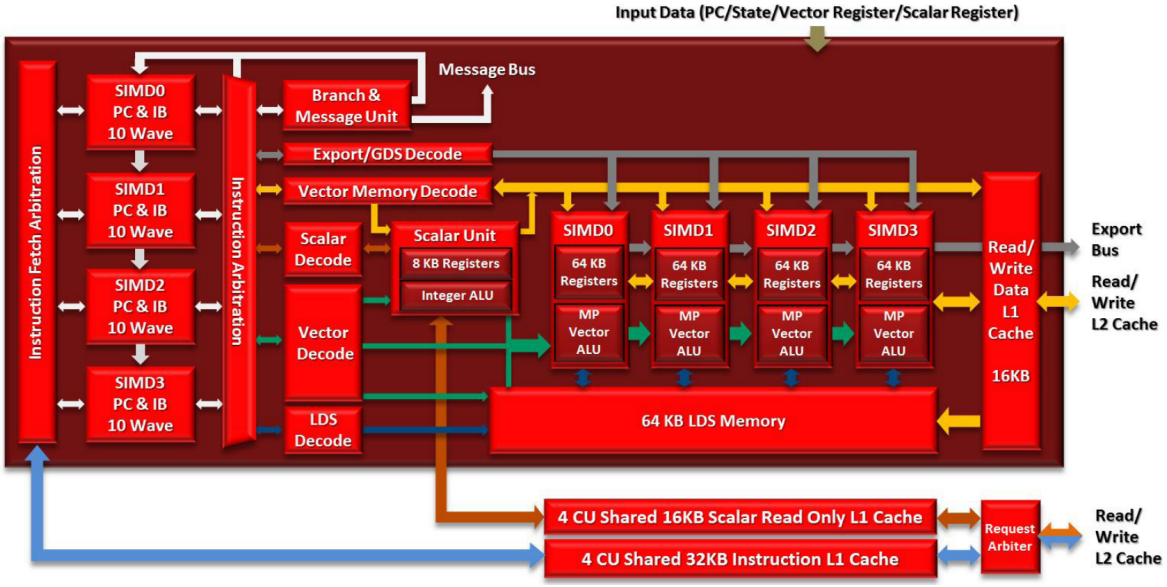


Figure I.14: Architectures GCN (AMD) : détail d'un *Compute Unit* (© AMD)

On notera cependant que, comme ce serait le cas pour n'importe quel type d'architecture, ces nombres de threads sont purement théoriques. En effet dans la pratique la définition du travail à réaliser par un thread est le fait du programmeur, qui doit considérer le problème à traiter ; et les caractéristiques dudit problème ne sont pas forcément en phase avec celles de l'architecture considérée.

HIÉRARCHIE MÉMOIRE : COHÉRENCE DES CACHES ET SYNCHRONISATION DES THREADS

L'accélérateur Radeon HD 7970 dispose de 3 Go de mémoire globale, rendue accessible aux unités de calcul *via* un système de caches hiérarchique :

- Le cache L2, global (figure I.13), est géré en 12 banques de cache L2 gérées par 6 contrôleurs mémoire *dual-channel* (partitions de 128 Ko chacune). Il est distribué mais on en a une vue globale par un système de cohérence de caches.
- Du cache L1 est disponible à deux niveaux différents (cluster et CU : voir figure I.14) :
  - cache de données L1 vectoriel de 16 Ko, en lecture/écriture, disponible au niveau des CU (pour alimenter les ALU vectorielles disponibles au sein des unités SIMD) ;
  - caches en lectures seule au niveau des clusters (donc pour un ensemble de 4 CU) : 32 Ko de cache L1 d'instruction, répartis en 4 banques ; 16 Ko de cache de données L1 scalaire, séparé, qui permet de stocker notamment les données à tester en vue des branchements par les unités de contrôle (limites de boucles, ...).

L'objectif de cette nouvelle structuration hiérarchique de la mémoire, rendue homogène au niveau L2 par son système de cohérence de caches, est de rendre l'architecture scalable lors d'une éventuelle future augmentation du nombre de clusters et/ou de CU

par cluster [10].

La collaboration entre les threads est assurée par deux espaces mémoire spécifiques, localement et globalement :
- Disponible au niveau de chaque CU, le LDS (*Local Data Share*) joue le rôle d'un troisième fichier des registres, étant utilisé particulièrement pour des synchronisations à l'intérieur d'un work-group. La capacité du LDS est de 64 Ko, gérés en 16 ou 32 banques, chacune ayant 512 entrées de 32 bits. Une SIMD peut charger ou stocker des données dans le LDS pour éviter la pollution des hiérarchies de caches par des accès *scatter* ou *gather*, et préserver la bande passante utilisable.
- Par ailleurs le GDS (*Global Data Share*) est partagé par l'ensemble des CU. Il sert comme un point de synchronisation global pour tous les wavefronts.

I.3.2 Accélérateurs spécialisés (ClearSpeed, CellBE, FPGA)

Les architectures GPU constituent des accélérateurs, qu'on peut utiliser pour décharger les processeurs classiques d'une partie de leur charge de calcul. Leur apparition est venue d'un besoin émanant du domaine de l'affichage video et leur utilisation en tant qu'accélérateurs de calcul n'est venue qu'ultérieurement. Néanmoins il existe d'autres types d'accélérateurs matériels de calcul qui, bien que pensés pour des domaines spécifiques, ont été explicitement conçus pour accélérer des calculs. Nous présentons ici les deux accélérateurs spécifiques le plus notables, à savoir les processeurs ClearSpeed et CellBE ; nous aborderons alors les circuits programmables FPGA, dont la caractéristique est justement que l'architecture est prévue pour être spécifiquement configurée en fonction des problèmes à traiter.

1) ClearSpeed - CSX700

ClearSpeed Technology est un concurrent direct de NVIDIA et AMD sur le marché des accélérateurs multicœurs. L'entreprise a lancé son premier processeur en 2005, devenant ainsi une des premières à proposer des accélérateurs dédiés au calcul à haute performance. L'atout majeur de ClearSpeed face à la concurrence figure dans l'efficacité énergétique de ses accélératuers, qui leur donne un avantage dans le domaine des systèmes embarqués. Leur dernier produit, le processeur CSX700, lancé en 2008, a une consommation de seulement 10 Watts pour 96 GFlops en double précision ; de plus à l'époque il s'agissait du seul produit disposant de la correction d'erreur (ECC).

Du point de vue de l'architecture le processeur inclut deux noyaux indépendants, MTAP (*Multi-Threaded SIMD Array Processors*), deux contrôleurs mémoire, ainsi qu'un contrôleur PCIe x16 (figure I.15). Le bus interne est connecté à l'extérieur par

10. Par ailleurs c'est cet espace mémoire virtuellement homogène, qui en simplifie la gestion, qui a permis à AMD de créer les processeurs hybrides CPU-GPU de la gamme AMD *Fusion* : appelés APU (*Accelerated Processing Units*), ils partagent un seul espace d'adressage.

le biais d'un *ClearConnect Bridge* (CCBR) qui permet de créer des systèmes *dual-chip* ou d'interfacer le processeur avec un contrôleur FPGA dans un système embarqué.

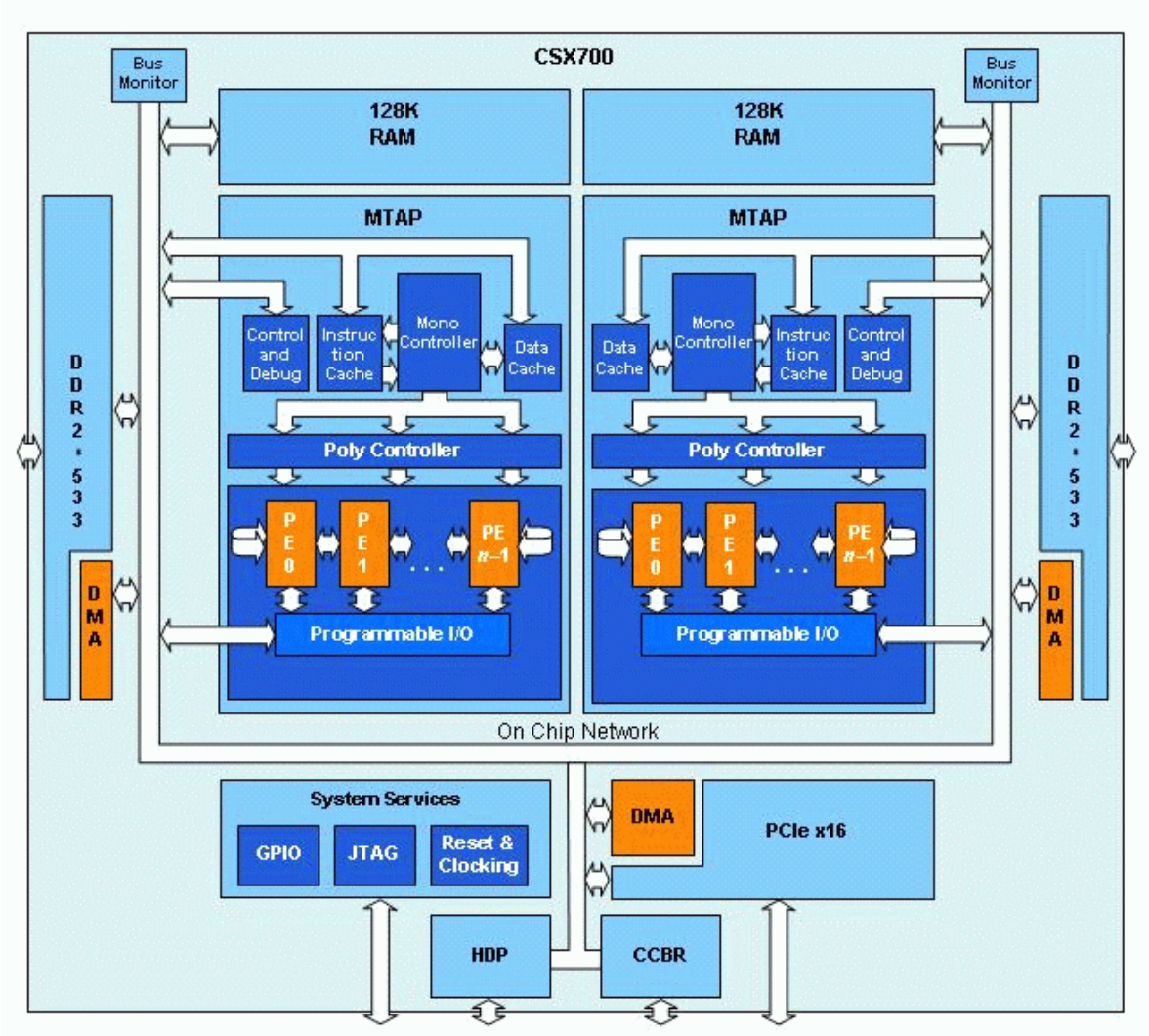


Figure I.15: L'architecture ClearSpeed (© ClearSpeed)

Les détails techniques sont présentés dans le tableau I.6 [Cle08].

L'architecture du MTAP (figure I.16 gauche) est une solution puissante et évolutive, basée sur un PEU (*Poly Execution Unit*), tableau d'éléments de traitement de type SIMD (PE, *Processing Elements*), auquel s'ajoute une unité dite MCU (*Mono Control Unit*) permettant de gérer les flots d'instructions et de données des différents PE. Le PEU de chaque MTAP est composé de 96 PE (92 actifs et 4 redondants), soit un total de 192 pour le processeur entier. Comme le montre la figure I.16 droite, chaque PE contient deux unités de calcul en virgule flottante en simple et double précision, une unité arithmétique et logique (ALU), une unité spéciale pour les divisions et les racines carrés, un accumulateur de 16 bits (MAC), 128 Ko de registres et 6 Ko de SRAM avec correction d'erreurs.

Caractéristique	Valeur
Cœurs	192 (96/MTAP) + 8 redondants en cas de panne
Mémoire SRAM embarquée	2 x 128 Ko
Mémoire externe	2 x 8 Go (DDR2)
Performance (crête) en simple et double précision	96 GFlops
Fréquence d'horloge	250MHz
Bande passante mémoire interne	192Go/s
Bande passante PCIe	4Go/s

TABLE I.6: Détails techniques du processeur ClearSpeed CSX700

L'architecture, et le jeu d'instructions identique pour les PEU et pour chacun des PE, permettent à l'ensemble du processeur d'être efficace à la fois pour du code séquentiel et pour du code parallèle.

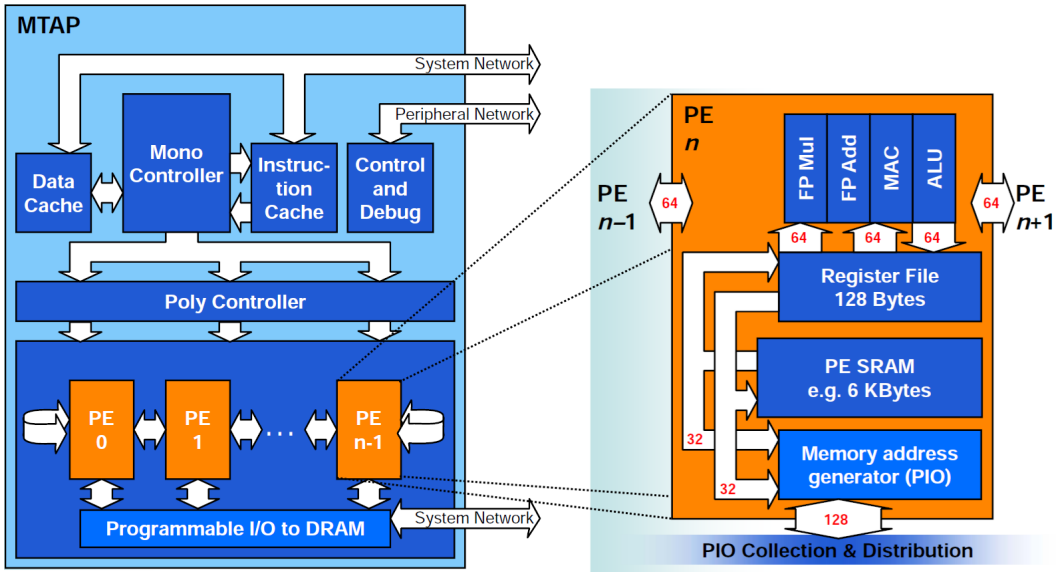


FIGURE I.16: Noyau MTAP ClearSpeed (© ClearSpeed)

Dans le cadre du partenariat PRACE, l'entreprise Petapath propose deux prototypes de supercalculateurs basées sur l'architecture ClearSpeed : le premier en partenariat avec SGI, hébergé au centre de calcul CINES, à Montpelier (France) et le deuxième en partenariat avec HP, hébergé au centre de calcul d'Amsterdam (Pays-Bas). Des retours de ce dernier montrent des accélérations allant jusqu'à 17 fois pour certains problèmes particulières, bien meilleures qu'avec la carte graphique NVIDIA Tesla utilisée pour les tests comparatifs.

2) IBM - *Cell Broadband Engine Architecture* (CellBE)

Le processeur *Cell Broadband Engine*, plus couramment nommé CBE ou encore Cell, est un microprocesseur qui se positionne dans la niche entre les processeurs classiques et les processeurs spécialisés de type cartes graphiques. Révélé en février 2005, ce processeur est le fruit du plusieurs années de travail du consortium STI, qui rassemble **S**ony Computer Entertainment, **T**oshiba Corporation et **I**BM [KDH+05]. Il équipe notamment la console de jeu vidéo PlayStation 3 de Sony, mais ses applications sont beaucoup plus larges :
- les jeux vidéo et les bornes d'arcade ;
- les applications multimédia, films, TVHD ;
- le rendu en temps réel, les simulations physiques, le traitement du signal ;
- l'imagerie médicale, l'aérospatial et la défense, le calcul sismique, les télécommunications.

Concernant l'architecture, plutôt que de dupliquer plusieurs fois le même cœur identique comme sur les processeurs multi-cœurs classiques, les concepteurs ont choisi une tout autre approche : considérer un cœur principal, et lui adjoindre 8 cœurs spécifiques (figure I.17). Ainsi un processeur CBE est composé de :
- 1 PowerPC Processing Element (PPE) ;
- 8 Synergistic Processing Elements (SPE) ;
- un cache de niveau 2 de 512 Ko, partagé ;
- l'*Element Interconnect Bus* (EIB) qui gère les communications internes entre les différents éléments ;
- le *Memory Interface Controler* (MIC) : contrôleur mémoire partagé ;
- le *RamBus Flex I/O interface* : contrôleur d'entrées/sorties ;
- le *Direct Memory Access Controller* (DMAC) ;
- 2 contrôleurs mémoire Rambus XDR.

Les données quantitatives sont présentées dans le tableau I.7.

Caractéristique	**Valeur**
Cœurs	1 PPE + 8 SPE
Mémoire SRAM embarquée	4 Mo (256 Ko / SPU)
Mémoire externe	8 Go
Performance (crête) en simple précision	230.4 GFlops
Performance (crête) en double précision	108.8 GFlops
Fréquence d'horloge	3.2 GHz
Bande passante mémoire	25 Go/s

TABLE I.7: Détails techniques du processeur PowerXCell 8i

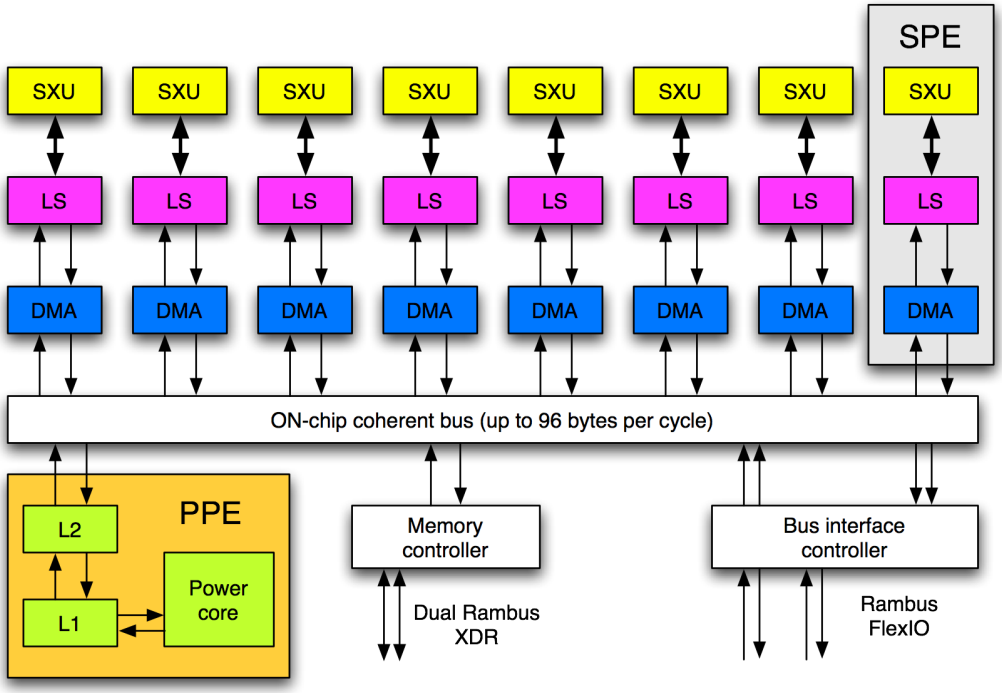


FIGURE I.17: L'architecture CellBE (© Hellisp)

PPE, *Power Processor Element* (figure I.18)

Le cœur principal, PPE, est relativement proche d'un cœur classique. Il est conçu pour exécuter le système d'exploitation et la plupart des applications, alors que les applications intensives seront déployés sur les SPE. Le PPE est dérivé d'une architecture Power 64 bit, dont il conserve le jeu d'instructions. Le processeur est *dual issue*, *dual threaded* et *in-order* .

SPE, *Synergistic Processing Elements* (figure I.19)

Chacun des 8 cœurs spécifiques, SPE, est constitué de deux parties : un contrôleur mémoire (MFC, *Memory Flow Controller*) et une unité de calcul vectoriel (SPU, *Synergistic Processing Unit*). Le processeur de type RISC est doté d'un jeu d'instructions SIMD spécifique mais se rapprochant de VMX (AltiVec). Les SPU ont un accès direct et extrêmement rapide à leur *Local Storage*, mémoire locale embarquée SRAM de 256 Ko. Ils intègrent aussi un espace de 128 registres de 128 octets chacun. Un SPU ne peut pas accéder directement à la mémoire principale, et doit pour cela effectuer une requête de transfert asynchrone à un bus d'interconnexion.

EIB, *Element Interconnect Bus*

L'anneau de communication, EIB, fait la connexion entre le processeur PPE, les huit coprocesseurs SPE, le contrôleur mémoire (MIC) et le deux interfaces d'entrées/sorties.

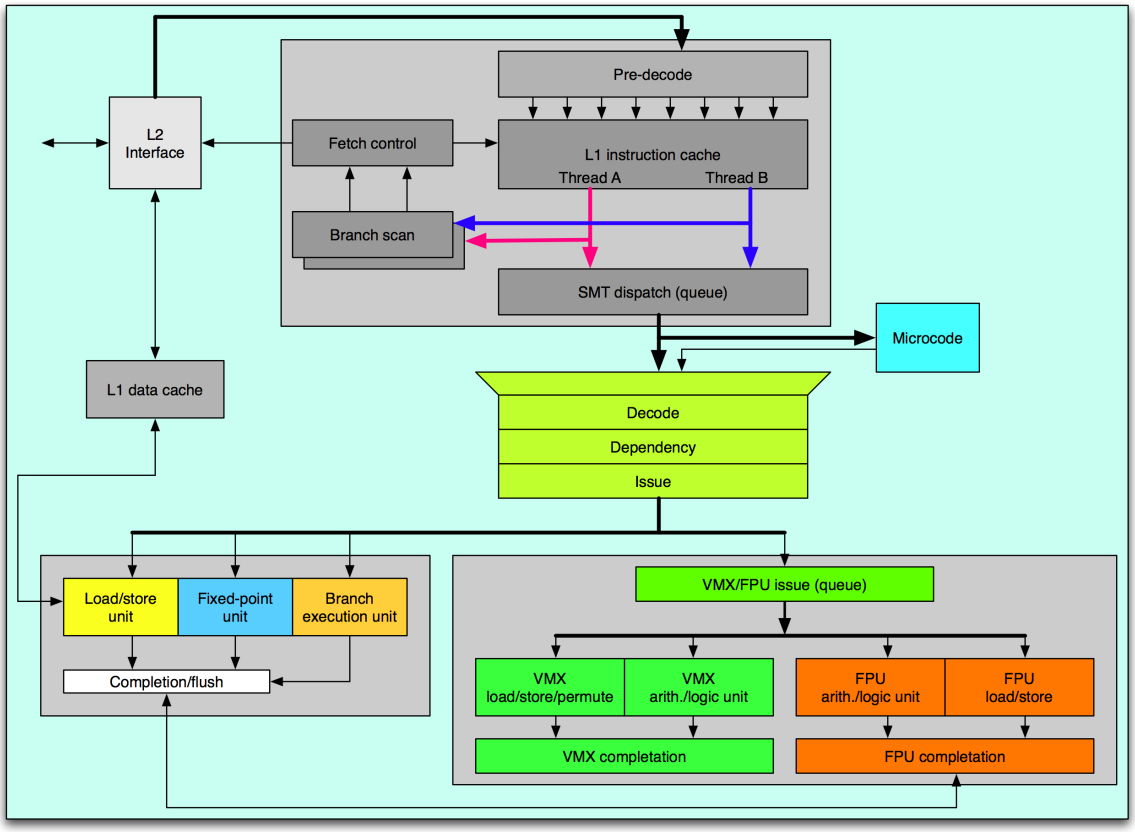


Figure I.18: Power Processor Element - CellBE (© Hellisp)

Si le processeur Cell a été initialement conçu pour les consoles de jeux, sa très grande puissance de calcul le rend très attractif dans le domaine du calcul intensif. Par contre sa complexité rend le portage d'applications très complexe car il faut optimiser les calculs et les transferts. La répartition des tâches sur les cellules SPE, ainsi que la communication entre les éléments, sont de vrais enjeux pour le programmeur qui souhaite utiliser au mieux ce processeur.
Le super-calculateur le plus puissant à utiliser des processeurs Cell est l'IBM Roadrunner. Ses 12 960 processeurs PowerXCell 8i, couplés avec 6 480 processeurs AMD Opteron (122 400 cœurs), l'ont fait entrer à la première position du Top500 en juin 2008 ; il a quitté cette première place 18 mois plus tard et est actuellement en 19ème position (juin 2012).

3) FPGA

Un FPGA (*Field-Programmable Gate Array*), circuit logique programmable ou réseau logique programmable, est un circuit intégré logique qui peut être reprogrammé après sa fabrication.

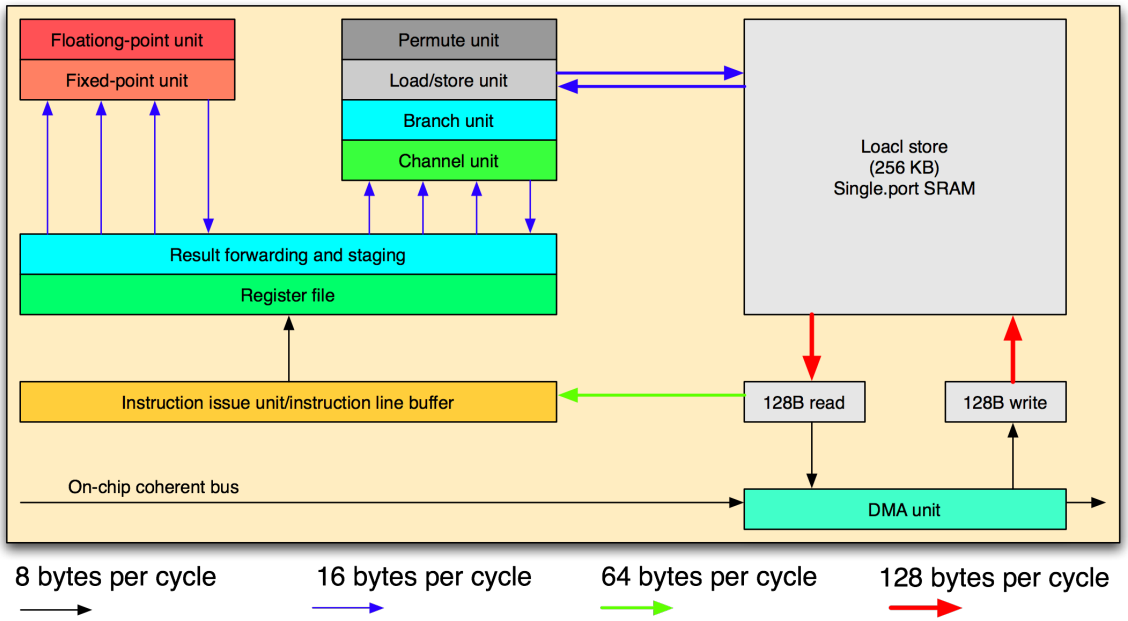


FIGURE I.19: Synergistic Processing Element - CellBE (© Hellisp)

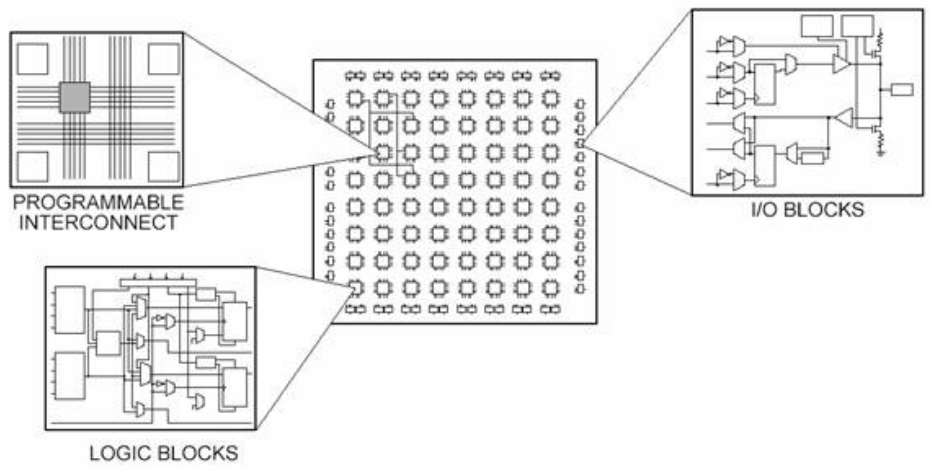


FIGURE I.20: Structure d'un FPGA : réseau d'interconnection programmable, block d'interconnection et bloc d'entrée/sortie (© National Instruments)

La plupart des grands FPGA modernes sont fondés sur des cellules SRAM aussi bien pour le routage du circuit que pour les blocs logiques à interconnecter (figure I.20). Un bloc logique est constitué d'une table de correspondance (LUT, *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 en 2007), sont connectés entre eux par une matrice de routage configurable [11]. Ceci permet la reconfiguration à volonté du composant, mais occupe une place importante sur le silicium et justifie le coût élevé des composants FPGA.

Pour illustrer l'architecture d'un FPGA nous présentons ici un des plus puissants modèles existants. Le Xilinx Virtex-7 XCV2000T est un des plus *gros* circuits intégrés actuellement commercialisés, intégrant 6,8 milliards de transistors en technologie 28 nm. Constitué de quatre puces incluant chacune plus de 10 000 connexions, ce circuit comprend près de deux millions de cellules logiques (1 954 560) pour des calculs massivement parallèles, 305 400 blocs logiques configurables, 21 500 Ko de mémoire RAM distribuée, 2 160 cellules DSP, 45,5 Mbits de mémoire RAM globale, 4 blocs PCIe et 36 émetteurs-récepteurs GTX chacun d'une vitesse de 12,5 Gbit/s, pour une consommation inférieure à 30 W (figure I.21).

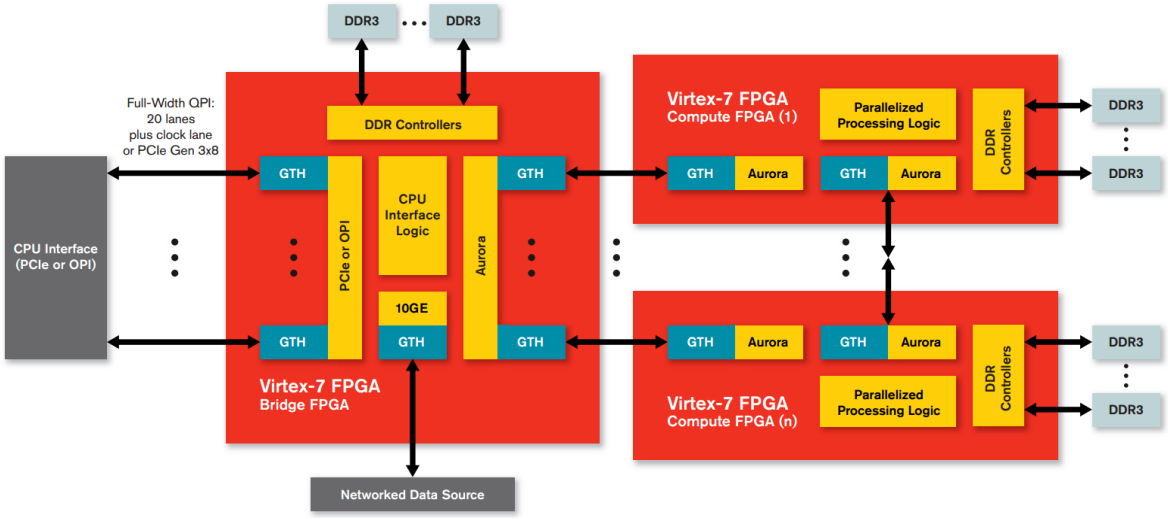


Figure I.21: L'architecture Xilinx Virtex-7 (© Xilinx)

I.3.3 L'avenir des architectures manycore ? (Intel - Xeon Phi)

Après l'abandon du projet Larrabee, en mars 2010 [SCS+08], Intel a proposé une nouvelle architecture manycœurs : MIC (*Many Integrated Core*) [Int12]. La disponibilité commerciale de cette architecture a été annoncée pour la fin de l'année 2012, sous le nom d'Intel Xeon Phi.

Intel présente ses cartes manycore comme une alternative aux accélérateurs actuels, d'une part en améliorant leur programmabilité (ils sont basés sur des architectures X86), et d'autre part en élargissant le cadre de leur utilisation [MDK12]. Sur ce second point, en effet, les plateformes de test permettent de fonctionner dans deux modes

11. La topologie est dite « Manhattan », en référence aux rues à angle droit de ce quartier de New York.

d'exécution différents :
– mode *offload* : comme pour les GPU, dans ce mode les parties de code marquées avec les directives appropriées sont automatiquement exécutées sur l'accélérateur ;
– mode *native* : comme pour un nœud multi-cœur, avec un système d'exploitation Linux embarqué, le programme s'exécute complètement sur le processeur.

Les premières versions de cette architecture, disponibles depuis courant 2010, sont nommées Knights Ferry (KF). Elles sont gravées dans la technologie de 45 nm tandis que les versions Knights Corner, prochainement disponibles, sont gravées en 22 nm.
Les caractéristiques techniques principales des modèles de test, Knights Ferry et Knights Corner, sont présentées dans le tableau I.8.

Caractéristique	**5110P**	**3100**
Nombre de cœurs	60	-
Fréquence d'horloge (MGHz)	1053	-
Performance (crête) en simple précision (GFlops)	-	-
Performance (crête) en double précision (GFlops)	1011	>1000
Taille mémoire (GB)	8	6
Bande passante mémoire (GB/sec)	320	240

TABLE I.8: Intel Xeon Phi : détails techniques

Les premières cartes disponibles commercialement ont été le 5110P en novembre 2012, alors que la serie 3100 à été annoncée pour courant 2013. Le modèle ayant le meilleur rendement est le 5110P. Il dispose de 60 cœurs et permet d'atteindre une puissance de 1,011 TFlops en double précision pour une consommation de 225 W avec refroidissement passif. La serie 3100 proposera plus de 1 TFlops en double précision pour une consommation aux alentours de 300 W.

L'architecture des Xeon Phi est composée d'unités SIMD de 16 éléments [12], reliées par une interconnexion en anneau sur 1024 bits (figure I.22). La puce Aubrey Isle utilisée est une version améliorée du processeur x86 Pentium 1 (P54C), auquel ont été rajoutées des unités vectorielles et une unité pour le calcul en double précision (figure I.23).
La version B0 de la puce existe visiblement dans plusieurs configurations, avec des nombres de cœurs différents (60 et 61). Ces configurations disposent de 1,8 à 1,9 Mo de mémoire cache L1 et de 28 à 30,5 Mo de mémoire cache L2. La mémoire principale est de 6 ou 8 Go de GDDR5.

12. Intel a confirmé qu'il proposerait des instructions SIMD de 512 bits, ce qui signifie que le processeur comportera des unités arithmétiques et logiques de 16 éléments.

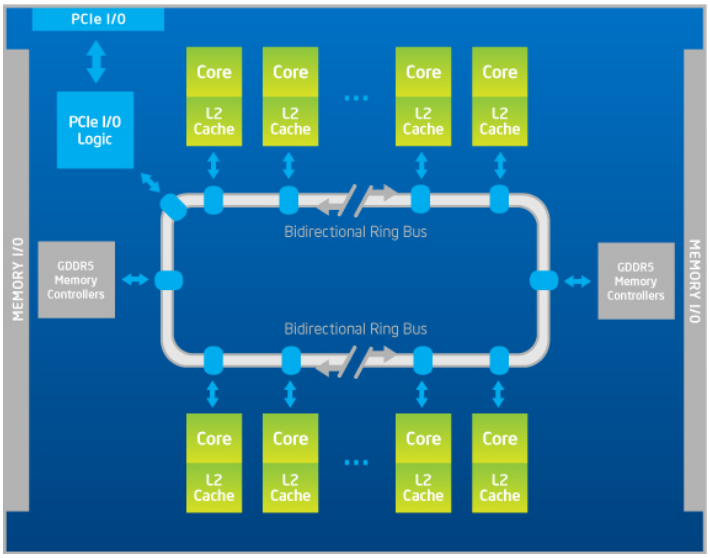


Figure I.22: L'architecture Xeon Phi (© Intel)

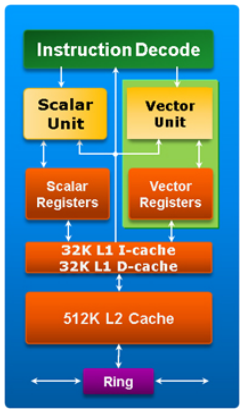


Figure I.23: Structure d'un cœur Xeon Phi (© Intel)

I.4 Conclusion

Dans ce chapitre nous avons présenté l'évolution du calcul parallèle. Dès le début, des classifications des différents types de modèles d'exécution ont été proposées. Au fil du temps de nouvelles architectures, disposant de milliers de cœurs de calcul, ont fait leur apparition dans ce paysage. Nous avons ainsi pu voir les accélérateurs matériels s'imposer dans le domaine du calcul parallèle, en étant utilisés par de plus en plus de supercalculateurs.
Nous avons donc présenté en détail ces accélérateurs matériels, du point de vue de l'architecture, pour mieux comprendre leur fonctionnement. Cela nous a en particulier

permis d'observer que, malgré de nombreux points communs [13], chacune de ces architectures a ses particularités ; une connaissance intime des architectures, en bas niveau, peut s'avérer indispensable pour les programmer et pour se rapprocher le plus possible de leurs performances crêtes annoncées.
Cependant l'hétérogénéité des architectures génère aussi une diversité des modèles de programmation. C'est dans cette optique que nous consacrons le prochain chapitre à l'étude des différentes approches de programmation des accélérateurs.

13. Parmi ces points communs on pointe notamment le nombre considérable d'unités de calcul et le respect du modèle SIMD, qui leur confère la dénomination d'architectures massivement parallèles.

II

Approches de programmation des accélérateurs

Sommaire

La diversité des architectures massivement parallèles entraîne une hétérogénéité des modèles de programmation. Mais avant même de pouvoir parler de programmation parallèle nous devons considerer les différents types de parallélisme, qui peuvent exploiter les différentes types d'architectures parallèles. Nous verrons alors que les modèles de programmation peuvent être de bas ou de haut niveau. Les langages de bas niveau comme CUDA ou OpenCL, sont fortement liés à l'architecture et difficiles à maîtriser ; c'est d'ailleurs pour cette raison que nous avons présenté les détails matériels des architectures dans le chapitre précédent. Cependant il existe des langages de plus haut niveau, qui sont plus faciles à utiliser et plus connus des programmeurs.

Parmi les langages de haut niveau, OpenMP se distingue depuis des années comme un standard *de facto* du calcul parallèle . Même si au début il n'a pas été conçu pour être utilisé avec des accélérateurs, car il a été initié bien avant que ceux-ci n'existent, ses idées novatrices (notamment l'utilisation des *pragma* sous forme de commentaires) sont encore d'actualité ; grâce à la facilité avec laquelle elles permettent d'exprimer le parallélisme elles sont reprises dans d'autre langages spécialement dédiés aux accélérateurs. Nous présentons trois langages basés sur des directives (PGI Accelerator, HMPP et OpenACC) ainsi qu'une brève comparaison.

Dans ce chapitre nous verrons aussi qu'avoir une large palette de modèles de programmation pose des problèmes non seulement pour le développement mais aussi pour la portabilité. Dans cette optique nous présenterons en fin de ce chapitre quelques transformateurs de code qui ont pour but de générer d'une manière automatisée du code capable d'être exécuté sur des cartes graphiques à partir de différents autres langages de programmation. Cette analyse nous permettra de positionner notre outil dans le cadre des outils de développement pour les processeurs graphiques.

II.1 Modèles de programmation parallèle

II.1.1 Les types de parallélisme

Le parallélisme, qui consiste à traiter des informations de manière simultanée, s'appuie principalement sur l'idée que les gros problèmes peuvent être divisés en des parties plus petites qui seront résolues en parallèle. Il y a plusieurs niveaux auxquels nous pouvons exprimer le parallélisme, que nous allons détailler ci-après par granularité croissante.

1) Parallélisme bas niveau

Historiquement, les processeurs 32 bits ont longtemps été les plus utilisés ; dans les dernières années ils ont été supplantés par les processeurs 64 bits, qui sont la norme aujourd'hui. Le parallélisme au niveau du bit (*bit-level parallelism*) est basé sur l'augmentation de la taille du mot mémoire. Cette méthode a été le moteur d'avancement dans les années 1970-1986, grâce à l'intégration à très grande échelle des composantes

(VLSI - *Very Large Scale Integration*). La vitesse de traitement des ordinateurs a pu être doublée par le doublement de la taille du mot, c'est-à-dire la quantité d'informations que le processeur peut manipuler par cycle.

2) Parallélisme d'instructions

Un programme correspond à un flot d'instructions exécuté par le processeur. Ces instructions peuvent être réordonnées ou regroupées pour être exécutées en parallèle sans changer le résultat du programme. Cela représente le parallélisme au niveau des instruction (*instruction-level parallelism*) Il y a deux approches complémentaires pour réaliser ce chevauchement d'instructions : au niveau matériel par une logique dédiée sur la puce ; au niveau logiciel grâce au compilateur. Des techniques variées au niveau de la micro-architecture permettent d'exploiter ce type de parallélisme :

- Le pipeline permet d'augmenter le débit des instructions en utilisant une séquence d'étages successifs pour découper l'exécution. Le plus souvent un pipeline comporte 5 étages : le chargement de l'instruction à exécuter (*Instruction Fetch*), le décodage de l'instruction et des adresses des registres (*Instruction Decode*), l'exécution par les unités arithmétiques et logiques (*Execute*), un transfert depuis un registre pour une instruction de type STORE ou vers un registre pour une instruction de type LOAD (*Memory*) ; le stockage du résultat dans un registre (*Write Back*).
- L'architecture superscalaire contient plusieurs pipelines en parallèle ; généralement chaque pipeline est spécialisé dans le traitement d'un certain type d'instruction.
- L'architecture superpipeline augmente largement le nombre d'étages, celui-ci pouvant par exemple aller jusqu'à 31 pour la microarchitecture Prescott et Cedar Mill d'Intel.
- Dans un autre domaine dans les architectures VLIW (*Very Long Instruction Word*) l'instruction va contenir les opérations pour chaque unité de calcul disponible dans le processeur, ce qui favorise du parallélisme entre les unités du processeur.
- Par ailleurs, l'exécution out-of-order consiste à réorganiser l'ordre dans lequel les instructions d'un programme vont s'exécuter pour essayer d'éviter les temps d'inactivité en exécutant les instructions immédiatement prêtes [1].
- Enfin, l'exécution spéculative est plus souvent utilisée pour la prédiction de branchement, qui constitue à repérer les instructions machine de branchement conditionnel à l'entrée du pipeline, et à prédire si le branchement est pris afin de charger le pipeline avec le bon flux d'instructions. Si la prédiction échoue, le pipeline doit être vidé et l'autre flux doit être chargé, ce qui occasionne une perte de temps proportionnelle à la longueur du pipeline. Pour autant, la qualité des prédictions de branchement permet d'obtenir de bons résultats.

Ce type de parallélisme est plutôt situé au niveau de l'architecture matérielle, sur laquelle le programmeur a peu ou aucun contrôle. Du point de vue du programmeur l'enjeu se trouve donc à des niveaux de parallélisme plus hauts.

1. Le procédé de sélection des instructions utilise l'algorithme de Tomasulo et le renommage des registres.

3) Parallélisme de données

Le parallélisme par distribution de données, ou parallélisme de données (*data parallelism*), est un paradigme de la programmation parallèle. Autrement dit c'est une manière particulière d'écrire des programmes pour des machines parallèles. Les algorithmes des programmes qui rentrent dans cette catégorie cherchent à distribuer les données au sein des processus et à y opérer les mêmes jeux d'opérations. C'est le cas lorsqu'on souhaite travailler sur des architectures SIMD.

Le parallélisme de données est exploitable sur des boucles du programme : les données sont distribuées à travers les unités de calcul différentes, pour être traitées en parallèle. Par contre pour pouvoir paralléliser une boucle il ne doit pas y avoir de dépendance entre les itérations de la boucle. De nombreuses applications scientifiques et techniques exploitent le parallélisme de données.

4) Parallélisme des tâches

Avec la granularité la plus forte, le parallélisme des tâches (*task parallelism*), encore appelé parallélisme fonctionnel ou de control, est un type de parallélisme dans lequel des processus différents sont distribués sur les nœuds de calcul et travaillent sur les mêmes données ou des données différentes.

Selon la taille des données à traiter par un seul processus, nous pouvons distinguer entre un parallélisme grossier (*coarse-grained parallelism*) et un parallélisme à grain fin (*fine-grained parallelism*).

On peut aussi distinguer deux catégories d'applications selon la quantité de communication entre les processus : applications parallèles couplées (*tightly coupled parallel applications*) ayant une importante quantité de communications inter-processus et applications parallèles indépendantes (*embarrassingly parallel applications*) ayant peu ou aucune communication, et qui se prêtent particulièrement bien à l'exécution parallèle.

Notre travail s'inscrit dans le contexte du parallélisme de données, exploité par les systèmes SIMD ; plus spécifiquement les cartes graphiques utilisent le parallélisme de données sous sa forme modifiée, SIMT. Par ailleurs leur architecture est pipelinée et permet donc de tirer profit du parallélisme d'instructions.

II.1.2 Langages de programmation

Il est possible de considérer les langages de programmation parallèle selon de nombreux points de vue [ST98, BST89] :
- selon qu'ils expriment le parallélisme explicitement (dans le code) ou implicitement (à la compilation/interprétation) ;
- en différenciant les langages de programmation des langages de coordination ;

– en opposant les langages « architecture-indépendants » et ceux qui sont liés à une architecture spécifique ; etc.

Par ailleurs, il faut différencier les langages de programmation parallèle selon leur conception : certains sont des extensions de langages séquentiels traditionnels (C, Fortran, Java) ou sont basés sur l'utilisation de librairies (PVM, MPI), alors que d'autres développent de nouveaux paradigmes de programmation, naturellement parallèles (programmation fonctionnelle, programmation logique, programmation objet, OCCAM/CSP).

Pour mener à bien la résolution des problèmes « en parallèle », les langages de programmation parallèle doivent être munis d'une possibilité d'échanger des données et des messages de synchronisation. Ces facilités, répertoriées sous le nom de communications inter-processus (*Inter-Process Communication*, IPC), regroupent un ensemble de mécanismes permettant à des processus concurrents (distants) de communiquer. Selon ce critère on envisage deux types de communications : en mémoire partagée ou par échange de messages. Ces modèles de communication correspondent aussi aux modèles de programmation parallèle de haut niveau correspondants : programmation en mémoire partagée (OpenMP, Pthreads) et par échange de messages (MPI). Nous les présentons ici globalement, avant de présenter les langages de programmation spécifiques pour accélérateurs graphiques.

1) Mémoire partagée

Dans les modèles de programmation à mémoire partagée, les tâches partagent un espace d'adressage commun de la mémoire ; elles y effectuent des accès (lectures/écritures) qui sont a priori asynchrones : ils doivent être contrôlés, par l'utilisation de verrous ou sémaphores. Le principal avantage des langages à mémoire partagée est qu'ils interdisent toute notion de propriété sur les données, et donc ne nécessitent pas de mécanismes compliqués pour échanger les données ; ils permettent d'en conserver une vue cohérente lorsqu'elles doivent être synchronisées. Les langages de programmation basés sur la mémoire partagée utilisent un modèle de processus légers (threads), chacun d'entre eux pouvant accéder à la mémoire partagée et disposant de mémoire privée. Ils exécutent leur code simultanément et de façon asynchrone (modèle BSP), et communiquent/se synchronisent via la mémoire globale.

Les threads sont gérés par un procédé de bifurcation-raccordement (*fork-join*). Ils agissent en parallèle sur des parties délimitées du code et il est donc possible de paralléliser les applications de façon incrémentale, ce qui facilite grandement leur mise-au-point : après avoir formulé le traitement en séquentiel et l'avoir éventuellement optimisé, on détermine les blocs d'instructions qui sont parallélisables indépendamment les uns des autres et on estime le gain qu'apporterait leur parallélisation, pour choisir lesquels seront paralléliser en premier.

Les **Threads POSIX** (ou PThreads) correspondent à une spécification IEEE de 1995.

Ils sont disponibles sur toutes les plate-formes UNIX. On les utilise par l'intermédiaire d'une librairie qui permet, en langage C, de décrire explicitement les relations précises entre les threads par des appels de fonctions : à leur création, il faut préciser quelles données ces processus partagent et leur attribuer un espace de mémoire propre, après quoi ils exécutent leurs codes respectifs de façon autonome (en préservant la cohérence des données par des blocages/déblocages de verrous). Tous les aspects parallèles de l'application doivent être programmés explicitement.

OpenMP, qui constitue une API de programmation en mémoire partagée, est également basé sur des threads. Développé avec l'appui de firmes industrielles et commerciales parmi les plus importantes dans les domaines matériel et logiciel (*OpenMP ARP Board*), il s'est imposé comme un standard de la programmation en mémoire partagée, et est porté sur toutes les plateformes UMA/CC-NUMA. Il est présenté plus en détail en II.3.1 : *OpenMP*.

2) Échange de messages

Le modèle de programmation par échange de messages est particulièrement adapté à une architecture sous-jacente de type NoRMA. Le réseau d'interconnexion permet la communication entre les processeurs, selon un certain routage : cela crée un canal de communication, éventuellement indirect, pour chaque paire de processeurs. Selon le modèle de programmation par échange de messages, une application est un ensemble de processus qui accèdent chacun à de la mémoire locale et se communiquent des données ou se synchronisent par des échanges de messages explicites (bloquants ou non bloquants). Ces processus sont répartis entre les processeurs. Le nombre de processus pour une application donnée est spécifié au début de l'exécution et ils sont actifs jusqu'à ce qu'elle prenne fin (en particulier il n'est pas possible d'avoir un nombre dynamique de processus). Les processus exécutent tous le même programme, mais il est possible de différencier leurs exécutions en testant leur index. Le premier avantage de la programmation par échange de messages est qu'il est relativement simple de débugguer une application conçue selon ce modèle car, chaque processus contrôlant sa propre mémoire, aucun ne peut modifier accidentellement une donnée contrôlée par un autre. Par ailleurs, la programmation par échange de messages n'est pas limitée aux architectures NoRMA, et s'adapte même à tous les types d'architectures répondant au modèle d'exécution MIMD. Sur les architectures UMA et NUMA, la mémoire est partagée mais les échanges de messages sont simulés par l'utilisation de variables partagées servant de buffers.

Historiquement, chaque constructeur a développé sa propre bibliothèque d'échange de messages pour permettre de compléter des codes réalisés dans des langages existants, mais sans qu'un standard puisse émerger. Dès 1989, certains laboratoires ont conjugué leurs efforts pour obtenir une librairie normalisée, PVM, mais la version publique n'a été disponible qu'en 1993. Pendant le même temps, les grands constructeurs et éditeurs de logiciels se sont rassemblés pour mettre au point un standard de programmation

par échange de messages, MPI (Message Passing Interface) : les bases ont été posées en 1992 et la première version date de 1994. La seconde version, finalisée 1997, est étendue au Fortran et au C++. Elle fait de MPI un standard industriel, qui a supplanté toutes les autres librairies de programmation par échange de messages (malgré l'impact de PVM). Plusieurs versions gratuites sont disponibles à ce jour : MPICH, OpenMPI, MVAPICH, etc.

Les modèles que nous venons de présenter s'intègrent dans les architectures MIMD, voire SIMD. Néanmoins les accélérateurs, et plus spécifiquement les cartes graphiques, sont des architectures à part, qui ne relèvent d'aucun de ces modèles. Pour cette raison de nouveaux langages de programmation bas niveau ont été développés pour ouvrir ces architectures à des calculs généralistes. Pour ces architectures on dispose d'un écosystème hétérogène de langages de programmation, parfois non-portables à cause du lien étroit avec le matériel. La section suivante présente les langages de programmation pour les cartes graphiques les plus utilisées en ce moment.

II.2 Langages de programmation spécifiques pour accélérateurs graphiques

Les deux langages propriétaires poussés en avant par les producteurs des cartes graphiques sont CUDA (pour NVIDIA) et C++ AMP (pour AMD). Il n'y aucune compatibilité, ni portabilité entre ces deux langages car ils sont étroitement liées aux architectures matérielles cibles. CUDA a une position dominante sur le marché car il était le premier, à son arrivée en 2007, à proposer d'ouvrir une architecture GPU à de calculs généralistes. L'initiative du consortium Kronos avec le modèle OpenCL a pour but de permettre de concevoir des programmes dont le code est indépendant de l'architecture cible, mais c'est en fait au détriment de la facilité de programmation. Cette partie présente en détail ces trois langages et met en évidence les particularités de chacun.

II.2.1 *CUDA* - NVIDIA

En février 2007, NVIDIA Corporation a rendu publique la version initiale du **Compute Unified Device Architecture *(CUDA)*** [KH10]. Le kit de développement CUDA permet au programmeur d'utiliser le langage de programmation C pour écrire des algorithmes capables d'être exécutés sur les cartes graphiques NVIDIA (depuis la version GeForce 8). CUDA permet aux développeurs un accès illimité aux architectures graphiques massivement parallèles des GPU NVIDIA, par l'intermédiaire d'un jeux d'instructions natif.

La partie hôte du runtime CUDA [NVIc] inclut des fonctions spécifiques permettant de spécifier les caractéristique de l'exécution pour manipuler :
– la gestion du périphérique
– la gestion des contextes

– la gestion mémoire
– le contrôle de l'exécution (appels des fonctions kernel)
– la gestion des textures
– l'interopérabilité avec les interfaces graphiques OpenGL et Direct3D

Le langage CUDA est composé de deux API qui correspondent à deux niveaux de complexité :

– *C runtime for CUDA API* est le plus couramment utilisé car il gère automatiquement la configuration des *kernels* avant leurs lancement, ainsi que leur lancement effectif. Il gère également des initialisations implicites de l'environnement CUDA, la gestion contextes et le passage de paramètres. Il est basé sur des extensions spécifiques du langage C99 ;
– *CUDA Driver API* est une API de bas niveau, qui offre un meilleur niveau de contrôle, avec l'inconvénient de nécessiter plus de code pour une même tâche et d'être plus difficile à programmer et déboguer. Il n'a aucune dépendance avec la bibliothèque d'exécution et n'a aucune des extensions C. Ces particularités permettent à d'autres compilateurs que le compilateur par défaut de NVIDIA d'être utilisé.

Le CUDA driver API est supporté par la librairie *nvcuda* et toutes ses fonctions sont préfixés par **cu**, tandis que le C runtime for CUDA API est supporté par la librairie *cudart* et toutes ses fonctions sont préfixés avec **cuda**.

Un exemple de kernel simple, pour le produit matriciel, est présenté ci-dessous.

Listing II.1: Produit matriciel - kernel CUDA

```
1  global void sharedABMultiply(float *a, float* b, float *c, int N)
2  { __shared__ float aTile[TILE_DIM][TILE_DIM];
3    __shared__ float bTile[TILE_DIM][TILE_DIM];
4    int row = blockIdx.y * blockDim.y + threadIdx.y;
5    int col = blockIdx.x * blockDim.x + threadIdx.x;
6    float sum = 0.0f;
7    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
8    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
9    __syncthreads();
10   for (int i = 0; i < TILE_DIM; i++)
11           sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
12   c[row*N+col] = sum;
13 }
```

Listing II.2: Produit matriciel - appel du kernel CUDA

```
1  // setup execution parameters
2  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
3  dim3 grid(WC / threads.x, HC / threads.y);
4  // execute the kernel
5  matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
```

La dernière version (5) disponible fin 2012, devrait apporter d'autres améliorations significatives :

– support pour le parallélisme dynamique, dans lequel chaque thread pouvait générer et lancer des sous-kernel, ce qui permet d'avoir de la récursivité ;
– technologie GPUDirect pour permettre la communication directe entre deux cartes graphiques ;
– l'édition des liens avec des librairies externes.

II.2.2 *Accelerated Parallel Processing* - AMD

AMD a rendu publique en décembre 2007 une première version de son kit de développement, première version **Stream Computing SDK**. Le kit de développement inclut « Brook+ », une version optimisée pour le matériel AMD.

Récemment, AMD a abandonné le langage Brook+ pour supporter pleinement le langage OpenCL présenté dans le chapitre suivant et a renommé l'environnement de développement **Accelerated Parallel Processing SDK**. De plus AMD, en partenariat avec Microsoft ont également annoncé un nouveau langage de programmation, C++ Accelerated Massive Parallelism (C++ AMP), qui en effet est une extension du langage C++. A ce jour, C++ AMD n'est pas supportée par le compilateur Microsoft Visual Studio disponible sur Microsoft Windows [Gre11].

Listing II.3: Produit matriciel - kernel C++ AMP (AMD)

```
template<typename _type, int tile_size>
void mxm_amp_tiled(int M, int N, int W, const std::vector<_type>& va,
    const std::vector<_type>& vb, std::vector<_type>& vresult)
{
    if ((va.size()!=M*N) || (vb.size()!=N*W) || (vresult.size()!=M*W))
        throw "Expected␣matrix␣dimension␣result(M*W)=a(MxN)*b(N*W)";

    extent<2> e_a(M, N), e_b(N, W), e_c(M, W);

    assert((M%tile_size) == 0);
    assert((W%tile_size) == 0);
    assert((N%tile_size) == 0);

    // Copy in
    array_view<const _type, 2> av_a(e_a, va);
    array_view<const _type, 2> av_b(e_b, vb);
    array_view<_type, 2> av_c(e_c, vresult);

    extent<2> compute_domain(e_c);

    parallel_for_each(compute_domain.tile<tile_size,tile_size>(), \\
        [=] (tiled_index<tile_size,tile_size> tidx) restrict(amp)
    {
        _type temp_c = 0;

        index<2> localIdx = tidx.local;
        index<2> globalIdx = tidx.global;

```

```
29          for (int i = 0; i < N; i += tile_size)
30          {
31              tile_static _type localB[tile_size][tile_size];
32              tile_static _type localA[tile_size][tile_size];
33
34              localA[localIdx[0]][localIdx[1]] =  \\
35                  av_a(globalIdx[0], i + localIdx[1]);
36              localB[localIdx[0]][localIdx[1]] =  \\
37                  av_b(i + localIdx[0], globalIdx[1]);
38
39              tidx.barrier.wait();
40
41              for (unsigned k = 0; k < tile_size; k++)
42              {
43                  temp_c += localA[localIdx[0]][k] *   \\
44                      localB[k][localIdx[1]];
45              }
46
47              tidx.barrier.wait();
48          }
49
50          av_c[tidx] = temp_c;
51      });
52      // copying out data is implicit - when array_view goes out of
53      // scope data is synchronized
54  }
```

Listing II.4: Produit matriciel - appel du kernel C++ AMP sur l'hote (AMD)

```
1       std::vector<DATA_TYPE> v_a(M * N);
2       std::vector<DATA_TYPE> v_b(N * W);
3       std::vector<DATA_TYPE> v_c_simple(M * W);
4       std::vector<DATA_TYPE> v_c_tiled(M * W);
5       std::vector<DATA_TYPE> v_ref(M * W);
6
7       initialize_array(v_a, M * N);
8       initialize_array(v_b, N * W);
9
10      mxm_amp_tiled<DATA_TYPE, 16>(M, N, W, v_a, v_b, v_c_tiled);
11
12      ...
```

II.2.3 *OpenCL* - Kronos

Apple, en collaboration avec AMD, Intel et NVIDIA, se sont réunis au sein du consortium Kronos et ils ont proposé les spécifications d'**OpenCL** (Open Computing Language) [Ope10]. L'enjeu était de proposer une API portable capable de prendre en charge tous les types d'accélérateurs matériels. La première version des spécifications OpenCL a été rendue publique en novembre 2008. La dernière version (1.2) contient

trois parties [ATI09] :
- spécification du langage (basé sur C99) et de l'interface de programmation (fichiers d'en-tête) permettent d'écrire des programmes portables, qui s'exécutent sur plusieurs plateformes hétérogènes constitué de processeurs, de GPU, et d'autres accélérateurs
- une couche API traite la gestion des périphériques permettant au développeur d'interroger la disponibilité des appareils et leurs caractéristiques, puis sélectionner et initialiser les dispositifs nécessaires aux calculs ; cette API traite également le contexte et la gestion des files d'attente
- l'interface d'exécution (runtime API) permet aux développeurs de créer des objets mémoire associés à des contextes, de compiler et de créer des objets kernel, appeler des commandes vers la file d'attente ou realiser des synchronisations et libérer des ressources utilisées.

Le modèle d'exécution d'OpenCL est similaire à celui du langage bas niveau de CUDA (*CUDA driver API*) mais oblige à utiliser, explicitement, des opérations telles que :
- la création d'éléments du contexte attaché au périphérique (collection de périphériques OpenCL utilisés par l'hôte), les kernels (fonctions OpenCL qui s'exécutent sur des périphériques), des objets programme (la source du programme et l'exécutable qui met en œuvre les kernels), des objets mémoire (ensemble d'objets visibles à l'hôte et aux dispositifs OpenCL) ;
- la création d'une file d'attente de commandes, qui inclue les instructions planifiées pour s'exécuter sur le périphérique dans le cadre du contexte (instructions d'appel de kernel, instructions mémoire, instructions de synchronisation) ; il est possible d'associer à un seul contexte plusieurs files d'attente indépendantes.

Listing II.5: Produit matriciel - kernel OpenCL

```
1  __kernel void
2  matrixMul( __global float* C, __global float* A, __global float* B,
3             __local float As[BLOCK_SIZE * BLOCK_SIZE],
4             __local float Bs[BLOCK_SIZE * BLOCK_SIZE])
5  {
6      // Block index
7      int bx = get_group_id(0);
8      int by = get_group_id(1);
9
10     // Thread index
11     int tx = get_local_id(0);
12     int ty = get_local_id(1);
13
14     // Index of the first sub-matrix of A processed by the block
15     int aBegin = WA * BLOCK_SIZE * by;
16
17     // Index of the last sub-matrix of A processed by the block
18     int aEnd   = aBegin + WA - 1;
19
20     // Step size used to iterate through the sub-matrices of A
21     int aStep  = BLOCK_SIZE;
22
23     // Index of the first sub-matrix of B processed by the block
```

```
24      int bBegin = BLOCK_SIZE * bx;
25
26      // Step size used to iterate through the sub-matrices of B
27      int bStep  = BLOCK_SIZE * WB;
28
29      // Csub is used to store the element of the block sub-matrix
30      // that is computed by the thread
31      float Csub = 0;
32
33      // Loop over all the sub-matrices of A and B
34      // required to compute the block sub-matrix
35      for (int a = aBegin, b = bBegin;
36               a <= aEnd;
37               a += aStep, b += bStep) {
38
39          // Load the matrices from device memory
40          // to shared memory; each thread loads
41          // one element of each matrix
42          AS(ty, tx) = A[a + WA * ty + tx];
43          BS(ty, tx) = B[b + WB * ty + tx];
44
45          // Synchronize to make sure the matrices are loaded
46          barrier(CLK_LOCAL_MEM_FENCE);
47
48          // Multiply the two matrices together;
49          // each thread computes one element
50          // of the block sub-matrix
51          for (int k = 0; k < BLOCK_SIZE; ++k)
52              Csub += AS(ty, k) * BS(k, tx);
53
54          // Synchronize to make sure that the preceding
55          // computation is done before loading two new
56          // sub-matrices of A and B in the next iteration
57          barrier(CLK_LOCAL_MEM_FENCE);
58      }
59
60      // Write the block sub-matrix to device memory;
61      // each thread writes one element
62      C[get_global_id(1)*get_global_size(0)+get_global_id(0)]=Csub;
63
64  }
```

Listing II.6: Produit matriciel - appel du kernel OpenCL

```
1  size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
2  size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, WC),
3         shrRoundUp(BLOCK_SIZE, workSize[0])};
4  clEnqueueNDRangeKernel(commandQueue[i],multiplicationKernel[i],2,0,
5         globalWorkSize, localWorkSize,0,NULL,&GPUExecution[i]);
```

Prise en charge du modèle OpenCL

– Le 20 avril 2009, NVIDIA a été le premier a annoncer qu'il supportait le pilote OpenCL en bêta et quelques semaines plus tard est devenu certifié par le consortium Khronos [ope09b].
– Le 5 août 2009, AMD a dévoilé les premiers outils de développement OpenCL dans le cadre de sa version 2.0 du kit de développement ; ils l'ont proposé comme une alternative à leur langage « Brook+ » [AMD10]. Aujourd'hui AMD a complètement abandonné ce dernier langage et soutient pleinement l'OpenCL.
– Le 28 août 2009, Apple a publié Mac OS X Snow Leopard, qui supporta pleinement OpenCL et qui l'utilise pour certaines tâches graphiques du système d'exploitation [ope09a].

II.3 Langages de haut niveau pour gérer les accélérateurs graphiques

Dans la section précédente nous avons vu que les langages de bas niveau ont tendance à être difficiles à acquérir car le programmeur doit décrire explicitement toutes les interactions avec le périphérique (initialisation du périphérique, transferts de données, initialisation des calculs, etc). Lorsque les GPU ont atteint leur masse critique, l'intérêt des entreprises pour ces architectures a initié une demande de modèles de programmation haut niveau adaptés pour en tirer parti. Nous allons présenter les principaux langages de haut niveau.

Dans le cas des langages haut niveau la pression est transférée du programmateur vers le compilateur, qui doit prendre en charge et générer automatiquement le code nécessaire en se basant sur des indications insérées dans le code (directives ou appels à des fonctions d'une bibliothèque). Le modèle de programmation OpenMP est devenu un standard *de facto* de la programmation en mémoire partagée. Nous débutons donc cette présentation avec le modèle OpenMP qui, nous le verrons dans les chapitres suivants, se prête également bien à la gestion multiGPU.

II.3.1 OpenMP

Un langage de programmation parallèle doit en principe donner la possibilité de décrire trois aspects du parallélisme : spécifier des régions qui seront exécutées en parallèle, gérer la communication entre plusieurs threads ou processus, ainsi que leur synchronisation.

OpenMP est une implémentation multithreadée de ces principes, basée sur un modèle d'exécution parallèle où un thread maître génère des threads esclaves pour que tous travaillent en parallèle [CDK+00, CJvdPK07]. Développé avec l'appui de firmes industrielles et commerciales parmi les plus importantes dans les domaines matériel et logiciel (*OpenMP ARP Board*), il s'est imposé comme un standard de la programmation en mémoire partagée. Il permet de paralléliser des applications écrites en C,

C++ ou Fortran en y ajoutant des directives de compilation (pour préciser les étapes de bifurcation-raccordement) ou des appels de routines de sa librairie (consultation du nombre de processus ou de l'index de l'un d'entre eux, positionnement de verrous, etc.). Après avoir préparé les threads esclaves, les parties parallèles du code peuvent être évoquées de deux façons différentes [Ope08] :

– Les boucles parallèles permettent de répartir les indices de boucles entre les processeurs selon des stratégies de répartition statiques ou dynamiques.
– Les régions parallèles font travailler les processus simultanément sur une même portion de code, qu'ils exécutent concurremment [2]. Au sein d'un région parallèle, une zone de code devant s'exécuter en exclusion mutuelle entre les processeurs sera par exemple isolée par une section critique.

Le figure II.1 présente d'une manière synthétique les directives OpenMP, avec les clauses correspondantes pour en préciser les conditions.

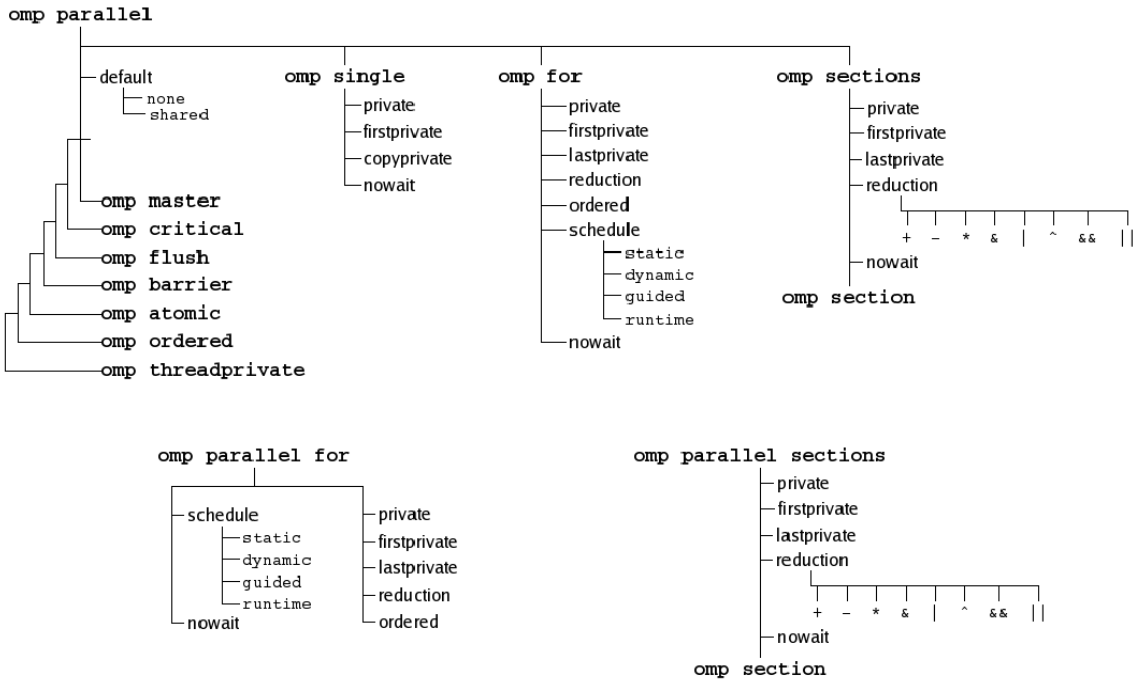


Figure II.1: Les directives et clauses OpenMP

La viabilité de ce modèle de programmation a été prouvée pendant des années et il reste pratiquement inchangé dans son principe, même si plusieurs révisions ont rajouté des fonctionnalités. Bien qu'il ne s'ágisse pas d'un processeur classique, l'accélérateur Intel Xeon Phi supportera le modèle OpenMP.

Comme il s'appuie sur les boucles parallèles ce modèle de programmation est particulièrement bien adapté pour exprimer le parallélisme de données en vigueur dans les architectures SIMT des accélérateurs graphiques. Nous considerons donc OpenMP

2. S'ils doivent exécuter des codes différents, on le spécifie en les différenciant selon leurs index

comme un langage de haut niveau pour la programmation sur GPU ; la génération de code bas niveau adapté est réalisé par notre transformateur de code, présenté dans le chapitre III : *Transformation automatique de code OpenMP en code CUDA*.

II.3.2 PGI Accelerator

Le modèle de programmation défini par PGI [3] permet de déporter des régions de calcul intensif sur le périphérique géré par l'hôte. Le périphérique peut exécuter des kernels, aussi simple qu'une boucle imbriquée ou un appel d'une routine complètement indépendante. Pour les régions ciblées, c'est l'hôte qui doit envoyer le code du kernel à l'accélérateur, orchestrer les allocations de mémoire sur le périphérique, initier un transfert de données et gérer la file d'attente, puis le transfert des résultats et la libération de la mémoire. Tous ces détails sont implicites et exprimés dans le modèle de programmation à l'aide de directives descriptives et sont gérées par le compilateur [Wol08]. Ainsi, les directives proposées ont ces deux finalités :

– délimiter des régions éligibles pour être exécutées sur le périphérique ;
– offrir des informations détaillées au compilateur pour faire un meilleur ordonnancement des boucles.

La directive de base est le `#pragma acc region`, pour définir une région du programme qui, après compilation pourra être exécutée sur le périphérique (voir le code du listing II.7). Les boucles à l'intérieur de ce bloc seront remplacées par des kernels. Des instructions supplémentaires pour les transferts des données vers et depuis l'accélérateur, seront générées directement dans du code assembleur. Le compilateur est capable de générer deux versions du code dans le même fichier binaire, l'une pour l'accélérateur et l'autre pour le hôte : en fonction de l'évaluation d'une conditionnelle le code pourra être exécuté sur l'un des deux supports [PG08].

La directive accepte des clauses pour augmenter l'information transmise au compilateur en précisant les données qui doivent être copiées depuis la mémoire de l'hôte vers la mémoire de l'accélérateur (`copyin`), et les résultats qui doivent être rapatries depuis l'accélérateur (`copyout`). La clause `local` est utilisée pour déclarer des données soit sur le hôte, soit sur l'accélérateur mais qui ne doivent pas être transférées.

Listing II.7: Produit matriciel - Exemple d'une région parallèle en PGI

```
1 #pragma acc region
2   {
3   for (j=0;j<m;j++)
4      for (k=0;k<p;k++)
5         for (i=0;i<n;i++)
6            a[i][j] = a[i][j] + b[i][k] * c[k][j];
7   }
```

3. PGI : *Portland Group, Inc.*

Au moment de la compilation des informations concernant les décisions prises pour la transformation sont affichées pour l'utilisateur dans un format spécifique (*Common Compiler Feedback Format*) les informations générés concernent :
- la façon de compiler s'il s'agit d'un appel d'une fonction ;
- des informations de profilage qui peuvent être utilisées dans les logiciels spécialisés (*PGI Profiler* ou *CUDA Profiler*) ;
- les optimisations appliquées par le compilateur : vectorisations, parallélisation, réordonnancement des boucles, génération des codes sur l'hôte et sur l'accélérateur, transferts mémoire ;
- une estimation de l'*intensité computationelle* d'une boucle, d'une région ou du programme, définie comme étant le rapport entre le nombre d'opérations exécutées sur un jeu de données et le nombre des transferts nécessaire pour déplacer les données.

II.3.3 HMPP

HMPP Workbench (*Hybrid Multicore Parallel Programming*) [DBB07] est une suite d'outils de développement, notamment un compilateur, pour la programmation hybride multi-cœurs et many-cœurs. Le standard OpenHMPP est basé sur le concept de *codelet*, qui est une fonction qu'il est possible d'exécuter à distance sur le matériel.

Globalement, le framework HMPP s'appuie sur :
- un jeu de directives (OpenHMPP), conçu pour manipuler les accélérateurs matériels sans se soucier de la complexité associée à la programmation GPU
- des générateurs de code qui extraient le parallélisme des codes C ou Fortran pour les transformer en kernels CUDA ou OpenCL
- une bibliothèque d'exécution qui gère la disponibilité, l'allocation et l'initialisation des accélérateurs, ainsi que l'arbitrage des communications entre l'hôte et l'accélérateur et l'exécution asynchrone des *codelets*.

Afin qu'il puisse être exécuté à distance, un *codelet* a les propriétés suivantes :
- c'est une fonction pure :
  - il ne contient pas de déclarations de variables statiques ou volatiles et ne fait aucune référence à des variables globales, exception faite des variables déclarées comme « resident » par une directive OpenHMPP
  - il ne contient pas d'appels à des fonctions avec un contenu non visible (c'est-à-dire qu'il ne peut pas être réalisé « inline ») : cela interdit donc l'utilisation de bibliothèques et les appels à des fonctions systèmes telles que `malloc`, `printf`, etc
  - chaque appel à la fonction doit faire référence à une fonction statique pure (pas de pointeurs de fonction)
- il ne retourne aucune valeur (l'équivalent en C est une fonction void ; l'équivalent en Fortran est une subroutine)
- le nombre d'arguments doit être fixé (pas de `vararg` comme en C).
- il n'est pas récursif
- ces paramètres sont supposés ne pas être aliasés (passage d'adresse, via un pointeur).

– il ne contient pas de directives *callsite* (c'est-à-dire pas d'appel RPC[4]) à un autre *codelet*) ou d'autres directives HMPP.

Ces propriétés permettent de s'assurer qu'un *codelet* RPC peut être exécuté à distance par un matériel. Cet appel RPC, et les transferts de données associés, peuvent être asynchrones.

La syntaxe générale des directives OpenHMPP (en langage C) est [Rao09] :

```
#pragma hmpp <grp_label> [codelet_label]? directive_type \\
    [,directive_parameters]* [&]
```

Un groupe des *codelets* est identifié d'une manière unique en utilisant le `grp_label`, tandis qu'un *codelet* peut être identifié par son `codelet_label`. Pour chaque *codelet* plusieurs versions peuvent être générées en fonction de l'architecture cible, la choix final étant réalisé au niveau de l'édition de liens.

II.3.4 OpenACC

En novembre 2011, CAPS, CRAY, NVIDIA et PGI se rejoignent au sein d'un consortium et fournissent la première spécification d'un langage basé sur des directives, nommé OpenACC, conçu explicitement pour les accélérateurs [Ope12]. Très similaire aux modèles OpenHMPP et PGI Accelerator, le nouveau standard a pour but d'uniformiser le paysage hétérogène des langages de haut niveau. Courant 2012, chacun des membres du consortium a sorti une version de son compilateur supportant les spécifications OpenACC 1.0, ayant comme architecture cible les GPU NVIDIA :

– CAPS supporte OpenACC à partir de la version 3.1 du HMPP Workbench ; les cartes graphiques AMD sont aussi supporté depuis juin 2012 et il y a des perspectives pour supporter les processeurs x86, les processeurs MIC et même Tegra 3 ;
– CRAY supporte OpenACC à partir de la version 8.0.6 du Cray Compiling Environment ; ce compilateur est spécialement conçu pour les supercalculateurs Cray XK6 ;
– PGI supporte OpenACC à partir de la version 12.6 du PGI Accelerator Compilers.

La syntaxe générale des directives OpenACC (en langage C) est :

```
#pragma acc directive-name [clause [[,] clause]?] new-line
```

Les fonctionnalités offertes par cette nouvelle API sont similaires à celles présentées précédemment :

– initialisation de l'accélérateur ;
– transferts implicites des données vers et depuis l'accélérateur ;
– coordination du travail sur l'accélérateur ;
– mappage les boucles parallèles dans le modèle d'exécution de l'accélérateur ;
– optimisation des opérations en utilisant des clauses pour affiner les informations disponibles à la destination du compilateur.

4. RPC : *Remote Procedure Call*

II.3.5 Comparaison

Comme nous venons de l'évoquer, la programmation basée sur des directives est le moyen le plus simple d'accéder à la puissance des accélérateurs avec un investissement minimum pour la modification de codes antérieurs. Sur cette base, les principaux acteurs du marché ont tenté d'imposer leur propre modèle de programmation, jusqu'à l'arrivée du OpenACC dont le but est de les unifier. Pour l'instant les modèles proposés par PGI Accelerator et OpenHMPP coexistent avec celui d'OpenACC, et les trois API continuent d'être développées en parallèle. Nous présentons donc ici une comparaison de ces trois langages de programmation basés sur des directives. Nous basons notre comparaison sur les deux points clés : d'une part le moyen d'exprimer le parallélisme dans les applications ; d'autre part la façon de spécifier les transferts mémoire et de les réaliser en pratique.

1) Expression du parallélisme

L'expression du parallélisme vient en amont de toute transformation technique des codes initiaux. Au niveau conception nous pouvons constater que OpenACC et PGI Accelerator sont très similaires et partagent le même concept de « zones » parallèles. Ces zones peuvent être soit des boucles soit des régions entières. De son coté OpenHMPP a comme unité de base le *codelet*, qui doit être une fonction. L'avantage des zones parallèles est le fait que n'importe quel programme peut facilement être parallélisé sans beaucoup de modifications alors que, dans le cas des *codelets*, le code doit être structuré en fonctions (s'il ne l'est pas encore, une réécriture s'impose). Néanmoins, l'avantage des *codelets* est que le code final peut disposer de plusieurs versions de la fonction : la version initiale pour une éventuelle utilisation sur CPU et celles transformées pour accélérateurs, avec éventuellement des versions différentes pour les différents types d'accélérateurs ; au moment de l'exécution l'environnement d'exécution peut choisir d'appeler l'une ou l'autre de ces versions, selon les ressources de calcul de la machine cible et éventuellement selon des expressions conditionnelles.

Au niveau des boucles parallèles, qui sont les zones de programme les plus intéressantes pour être accélérées, les approches sont également différentes :
- OpenHMPP supporte les boucles parallèles à l'aide de la directive spécialisée *hmppcg*, qui permet de donner des informations supplémentaires au compilateur, mais l'unité de base reste le *codelet* et les boucles sont des sous-parties d'un *codelet*.
- Dans le cas d'OpenACC et PGI Accelerator, il est possible de générer un kernel directement à partir d'une boucle et son appel sera implicite, alors que dans OpenHMPP les appels sont explicitement réalisées *via* des *callsites*.

2) Transferts mémoire entre l'hôte et le périphérique

Les trois langages disposent de directives spécifiques pour gérer les transferts, mais il y a toutefois des particularités pour chacun.

- OpenACC et PGI Accelerator sont capables de définir des régions de données dans lesquelles ils supportent les clauses `copyin`, `copyout` et `copy` pour gérer l'allocation mémoire sur l'accélérateur et les transferts vers/depuis l'accélérateur (ou dans les deux sens). Chaque modèle permet de faire des allocations sans aucun transfert (clause `create` pour OpenACC ; clause `local` avec PGI Accelerator).
- Les transferts de données constituant un goulot d'étranglement pour la performance, l'idéal est de réduire ces transferts et de réutiliser les données si elles ont déjà été transférées. Le modèle PGI offre deux attributs pour faire cela : `reflected` pour les arguments des procédures et `mirror` pour les données globales. De son côté OpenACC réduit cela à un seul attribut, `present`. L'avantage est l'intégration plus facile des appels à des bibliothèques externes.
- On peut donc limiter les transferts lorsque les données sont déjà sur l'accélérateur, en réalisant des tests de localisation. Pour éviter de tels tests, l'API OpenACC définit également quatre clauses de données combinées, associant la clause `present` avec respectivement `copyin`, `copyout`, `copy` ou `create`[5]. Lorsque les données sont déjà présentes sur l'accélérateur le comportement est celui de la clause `present` : les données sur l'accélérateur sont utilisés sans copie ; si les données ne sont pas présentes, alors elles sont allouées et copiées, comme l'aurait spécifié la clause correspondante (`copyin` / ...).
- De son coté OpenHMPP offre une fonctionnalité spécifique pour pré-charger les données sur l'accélérateur (directive `advancedload`) ou retarder leur rapatriement après calcul (directive `delegatedstore`). Grâce à cette possibilité de pré-chargement il est possible de spécifier, au moment de l'appel d'un kernel, le fait que les données sont déjà chargées dans la mémoire de l'accélérateur (clause `resident`, équivalente à la clause `persistent` d'OpenACC).

En plus de cela HMPP permet de réserver des accélérateurs (en mode exclusif ou non) et de créer des contextes associés. Cette fonctionnalité, qui n'a pas d'équivalent avec OpenACC, permet de faire en amont une pré-réservation de la carte graphique, sans avoir besoin de passer par une initialisation coûteuse au moment du lancement du *codelet* sur l'accélérateur. De même, une fois les *codelets* exécutés sur l'accélérateur, le programmeur peut décider de libérer ou non la ressource.

II.3.6 OpenMP for Accelerators

Toujours dans l'esprit d'utiliser des directives pour supporter la programmation des accélérateurs [ABB+10], un sous-comité d'OpenMP a été crée pour définir une exten-

5. Les clauses induites peuvent être évoquées dans leur forme longue (`present_or_copyin`, ...) ou courte (`pcopyin`, ...).

sion d'OpenMP appelée *OpenMP for Accelerators* [BSHdS11]. Les entreprises qui ont déjà défini le modèle OpenACC (PGI, CAPS, Cray) font partie de ce comité, qui doit proposer une première version d'OpenMP for Accelerators vers la fin de l'année 2012. Cela fait penser à certaines personnes qu'on peut considérer OpenACC comme une version bêta du futur modèle OpenMP entendu.

II.4 Transformateurs de code

Les compilateurs sont des programmes qui traduisent le code source, écrit dans un langage de haut niveau d'abstraction, facilement compréhensible par l'humain, vers un langage de bas niveau, généralement sous forme binaire (code objet ou *bytecode*). Pour la totalité des langages présentés précédemment, des compilateurs adaptés génèrent effectivement des codes binaires qu'on peut exécuter directement sur les architectures cibles.

Il y a une classe particulière de compilateurs, appelés transformateurs qui, à partir d'un code source initial, sont capables de générer, un autre code source équivalent dans un autre langage, par le biais d'une représentation intermédiaire. Ce processus est intéressant lorsqu'on dispose de langages de haut niveau pour exprimer les traitements à réaliser et lorsqu'on souhaite disposer d'un code source de plus bas niveau alors qu'il est difficile à écrire directement, ou lorsqu'on ne souhaite pas avoir à réaliser l'investissement de se former pour écrire directement les codes dans le langage de bas niveau.

II.4.1 Génération de code parallèle

Le tableau II.1 présente une brève comparaison de transformateurs de code significatifs. Ils sont décrits ici selon le critère des langages supportés, en entrée et en sortie. Nous n'y présentons pas les transformateurs permettant de générer du code pour accélérateurs graphiques, car ils sont présentés en détail dans la suite de cette partie.

Parmi ces transformateurs de code nous avons retenu l'OMPi pour sa simplicité et la possibilité de l'adapter facilement à nos besoins. Ce compilateur est capable de générer du code PThreads à partir du code OpenMP et sera présenté en détail en III.1. Tout le chapitre III : *Transformation automatique de code OpenMP en code CUDA* sera dédié a présenter les modifications que nous y avons apportées afin de générer du code CUDA à la place du code PThreads.

Il existe aussi des outils capables de générer du code spécifique pour les accélérateurs. On pense notamment à Cetus, Par4All ou encore KernelGen [ker12a], qui sont des outils récent développées pour générer du code CUDA. La suite de cette partie est consacrée à une brève présentation des caractéristiques de chacun d'entre eux.

	ROSE [LQPdS10]	LLVM [LA04]	OMPi [DLT03]	MPC Framework [CPJ10]	PoCC [BPCB10]
Code en entrée	ANSI C, C99, C++, Haskell, Fortran, Unified Parallel C, OpenMP (C/Fortran)	C, Fortran	OpenMP (C)	MPI (C/Fortran), OpenMP (C/Fortran), C+pragmas MIC	C
Code en sortie	C, Fortran, x64-86 ASM, code binaire	ASM, code binaire	C Pthreads	C	C+OpenMP optimizé, code binaire

TABLE II.1: Comparaison des principaux outils de transformation de code parallèle.

II.4.2 Génération de code pour accélérateurs

1) Cetus

Cetus [LME09] est un framework développé par l'université Purdue, qui permet de créer des traducteurs de code, source à source[6] [cet09], qui travaille à la transformation de code source écrit en langage C. L'architecture de compilateur Cetus est composé de : un analyseur C intégré (ANTLR), un générateur des représentations intermédiaires (RI) et plusieurs étages de modification du code. Le *front-end* permet d'intégrer plusieurs analyseurs car cet étage est séparé des étages de représentation intermédiaire.

Les étapes d'analyse comprennent :
– l'analyse des pointeurs
– l'analyse des éléments symboliques
– la privatisation des tableaux
– la reconnaissance des réductions
– l'analyse des dépendances de données
– la manipulation des expressions symboliques
– la génération des graphes d'appel et graphes de flot de contrôle
Les étapes de transformation comprennent : la substitution de variables, la parallélisation des boucles et la normalisation du programme.

Récemment, le compilateur a été enrichi avec un système de transformation OpenMP

6. *source-to-source code translator*

pour le GPGPU, composé de deux phases : l'optimiseur de flux OpenMP et la transformation de base OpenMP vers GPGPU (O_2G) [LME09]. Le traducteur utilise un algorithme pour identifier les régions qui seront transformées en des kernels, en s'appuyant sur les points d'inflexions qui constituent les instructions de synchronisation : `omp barrier`, `omp flush`, `omp critical`. Ensuite, pour le partitionnement de travail, les régions identifiées sont remplacées par des appels de kernel CUDA. La répartition des données est réalisée en s'appuyant sur des transferts mémoire adaptés.

Les optimisations du compilateur sont réparties à deux niveaux :
– les régions OpenMP, qui sont optimisées par rapport à la localité des données en utilisant soit des inversions de boucles pour les applications régulières, soit des fusions des boucles pour les applications irrégulières ;
– les optimisations spécifiques à la hiérarchie mémoire : elle comprennent la mise en cache des données fréquemment consultées, la transposition des matrices et la réduction des temps de transfert.

II.4.3 Par4All

Par4All [ACC+96] est une platforme développée par HPC Project avec le but de faire des migrations du code vers des architectures multicœurs et des architectures parallèles. Le projet s'appuie sur le framework PIPS (*Interprocedural Parallelizer of Scientific Programs*) qui est un transformateur du code capable d'analyser du code C ou Fortran séquentiel. La dernière version (1.4) du projet Par4All est capable de générer du code OpenMP, CUDA ou OpenCL. Le but principal du projet n'est pas essentiellement d'optimiser la performance, mais plutôt la réduction du temps nécessaire pour rendre une application disponible sur le marché, ainsi que la prise en main facile du code par un néophyte du domaine des GPUs.

La transformation du code séquentiel est réalisée en plusieurs étapes :
– les accès mémoire sont décrits comme des régions de lecture/écriture, qui ultérieurement serviront a générer des appels de gestion mémoire pour allouer ou libérer des ressources sur l'hôte ;
– une analyse de l'intégralité du code permet de déterminer les boucles qui peuvent être fusionnées ;
– la génération du code inclut des appels à une bibliothèque spécialisée pour les appels de fonctions GPU, pour faciliter la prise en charge des particularités du langage cible.

II.4.4 KernelGen

KernelGen [ker12b] est un traducteur source-à-source capable de paralléliser des boucles. Il peut générer du code GPU à partir de n'importe quel langage tant que celui-ci est supporté par le *front-end* GCC. L'approche de l'outil c'est de considérer le périphérique comme le système principal de calcul et d'y exécuter toutes les tâches, même celles qui doivent rester séquentielles (GPU active/Hôte passive). Le projet s'appuie sur le compilateur LLVM, plus particulièrement sur sa représentation intermédiaire plutôt que

sur l'arbre syntaxique.

L'image binaire générée par le *C Backend* contient à la fois le code hôte complètement fonctionnel et la version GPU, qui peut être activée à la demande. Dans la version GPU presque tout le code est implémenté sur le GPU, à quelques rares exceptions près, comme les appels système ou les fonctions issues des bibliothèques externes. Les données sont stockées entièrement sur le GPU et elles sont rapatriées sur l'hôte seulement à la demande.

II.5 Conclusions

Dans ce chapitre nous avons présenté les différentes approches de programmation des accélérateurs. Nous nous sommes concentrés sur les langages orientés vers les cartes graphiques, mais beaucoup de ces langages ou environnements de programmation peuvent être utilisés sans perdre de généralité pour d'autre types d'accélérateurs. Nous avons vu qu'il y a deux approches qui se dégagent pour la programmation des cartes graphiques. Il y a d'une part des langages de bas niveau pour une liaison étroite avec l'architecture cible, qui sont capables de tirer un maximum de performance du matériel (voir le chapitre précédent pour une présentation détaillée des architectures des accélérateurs). D'autre part il y a des langages de haut niveau permettant une programmation rapide et plus facile, pour des accélérations qui ne sont pas toujours les meilleures, mais qui donnent un retour rapide permettant de revenir ensuite sur le potentiel d'amélioration des applications. Ces derniers langages sont extrêmement utiles pour le milieu industriel où on cherche à avoir des résultats prometteurs, avec un minimum d'investissement, ainsi qu'un délai de mise sur le marché plus court. Nous avons pu constater que la plupart de ces langages de haut niveau sont basés sur l'utilisation de directives pour exprimer le parallélisme, conformément au modèle de programmation introduit par OpenMP depuis 1997. Cette approche est devenue un standard de facto de la programmation parallèle, et est supporté par la grande majorité des compilateurs actuels. Cette méthode de parallélisation, peu intrusive dans le code source, facilite la tache du programmeur mais déplace la difficulté vers le compilateur qui, à partir des indications des directives, doit être capable de générer par lui-même le code parallèle.

Grâce à sa maturité et du fait qu'il a prouvé son efficacité au fils du temps, OpenMP se présente aussi comme un bon langage de départ pour les transformateurs de code. Il y a de milliers de lignes de code qui profitent déjà de la parallélisation par des directives. Pour aider les programmeurs plusieurs projets ont été proposés, parmi lesquels certains sont capables de générer du code pour les accélérateurs à partir de code OpenMP. Notre thèse s'inscrit dans cette démarche, avec la contrainte principale de générer du code lisible par un humain, en vue d'optimisations ultérieures, notre approche sera détaillée dans le prochain chapitre.

III

Transformation automatique de code OpenMP en code CUDA

Sommaire

Nous avons vu dans les chapitres précédents qu'il y un lien très étroit entre l'architecture matérielle des accélérateurs et les différentes approches de programmation. L'avenir de l'exascale sera basé sur l'utilisation d'architectures massivement parallèles, vraisemblablement concues à partir d'accélérateurs matériels, et sur la possibilité d'écrire du code parallèle sans avoir à se soucier de l'architecture sous-jacente. Une des approches les plus naturelles pour tirer profit de la performance de ces nouvelles architectures est la programmation par directives ; elle permet notamment une parallélisation incrémentale des codes séquentiels avec un investissement limité à apporter sur la transformation du code.

Ce chapitre est consacré à notre principale contribution dans le cadre de la thèse : la description d'un environnement parallèle de développement haut niveau pour des accélérateurs graphiques capable de transformer du code OpenMP en du code CUDA. Comme nous l'avons déjà noté précédemment, OpenMP a été un choix logique du fait qu'il est devenu un standard de la programmation parallèle et qu'il existe beaucoup d'anciens codes qui reposent sur ce modèle de programmation. Notre outil est capable de prendre en charge une sous-partie du modèle OpenMP, notamment les boucles parallèles, qui se prêtent bien à une accélération sur les cartes graphiques. En sortie, nous avons choisi de générer du code CUDA, utilisé pour programmer les cartes graphiques NVIDIA, qui sont actuellement les plus répandues pour le GPU Computing et dans le domaine du HPC.

En pratique il s'agit essentiellement d'implémenter les transformations nécessaires pour générer un code CUDA à partir d'une boucle `pour` parallèle OpenMP. La contrainte principale que nous nous sommes fixée dans le cadre de cette thèse est que le code généré soit lisible par un humain, afin de permettre des modifications ou optimisations ultérieures. Dans le cadre de cette démarche plusieurs étapes de transformation sont nécessaires :

- le découpage de la structure composée « pragma omp parallel for » en deux expressions simples prenant chacune en charge une partie des variables (le pragma « omp parallel » prend toutes les variables sauf celles déclarées `private` ou `firstprivate`, qui sont rattachées au pragma « omp for »)
- le pragma « omp parallel » est transformé en un kernel CUDA, qui contient le corps du code rattaché initialement au pragma ainsi que toutes ses variables, qui sont transmises comme paramètres ; dans le code principal le pragma est remplacé par un bloc de code qui prend en charge l'appel du kernel, ainsi que les transferts mémoire nécessaires pour gérer les données dans la mémoire du périphérique
- le pragma « omp for » est pris en charge par le modèle d'exécution CUDA où un nombre important des threads GPU s'exécutent en parallèle (le corps du pragma est exécuté par chaque thread). Le nombre des threads est généré de façon automatique à partir des informations extraites de la boucle `pour` (limite inférieure, limite supérieure, pas d'incrémentation) et caractéristiques de l'architecture cible.
- la visibilité des variables est traitée au cas par cas : les variables partagées OpenMP se voient allouer des copies dans la mémoire globale du périphérique et les variables privées OpenMP se voient allouer sous forme d'un tableau dans la mémoire globale

(l'accès de chaque thread à sa copie se fait d'une manière optimale *via* des sous tableaux copiés dans la mémoire partagée du périphérique).

Dans le cadre de ce chapitre nous présentons en détail ces étapes de transformation, ainsi que des optimisations envisageables pour l'amélioration de notre outil. Notre transformateur de code est dédié à la génération de code CUDA, mais nous verrons que les transformations sont assez génériques pour permettre une évolution facile de l'outil et la prise en charge d'autres langages de programmation pour accélérateurs. Cette approche a donné lieu à une publication, présentée dans le cadre de la conférence « IEEE 13th International Conference on High Performance Computing and Communications » : HPCC2011 à Banff, Canada [NJK11].

III.1 Présentation du transformateur OMPi

Notre outil de transformation de code est basé sur OMPi [1], outil libre de droit développé par Parallel Processing Group à l'Université Ioannina en Grèce depuis 2001 [DLT03]. Dans cette section nous présentons OMPi et ses principales caractéristiques. Par la suite nous verrons comment l'adapter pour prendre en charge les spécificités des codes CUDA et nous décrirons les mécanismes utilisées à la transformation de boucles parallèles OpenMP.

OMPi est une implémentation portable de l'API OpenMP version 3.0 pour C. Le transformateur C vers C prend en charge du code C enrichi de directive OpenMP et produit du code C équivalent, en utilisant des threads POSIX. L'approche basée sur des Pthreads assure la portabilité du code généré. Le compilateur est entièrement implémenté en C, en contraste avec d'autres outils similaires comme OdinMP [KB04] ou Omni [SHI00], qui contiennent certaines parties implémentées en Java. Le choix d'implémenter le transformateur de code entièrement en C est dicté par l'objectif de performance de la compilation.

Les critères sur lesquels nous avons basé notre choix, et auxquels OMPi à répondu au mieux sont :

- la simplicité de la grammaire pour pouvoir intégrer les éléments spécifiques du langage CUDA ;
- la représentation intermédiaire du code, à la fois simple mais aussi complète et structurée, sans perdre la structure du code initial ;
- la re-génération d'un code lisible à partir de la représentation intermédiaire.

Les sections suivantes présenteront en détail le mode de fonctionnement, ainsi que des détails de son implémentation indispensables pour l'introduction des modifications apportées par nos soins pour la génération du code CUDA.

1. www.cs.uoi.gr/ ompi

III.1.1 Mode de fonctionnement

Le processus de compilation d'OMPi comporte plusieurs étapes de transformations du code source, pour produire le fichier final pour une exécution multithreadée. Essentiellement, une région parallèle est transformée de la façon suivante :
- le code de la région parallèle est déplacé dans une nouvelle fonction qui sera appelée plus tard par chaque thread ;
- le code initial est remplacé par une portion de code permettant de générer les threads qui appelleront la nouvelle fonction ;
- les variables privées sont redéclarées à l'intérieur de la nouvelle fonction ;
- les variables partagées, qui ont une portée globale, ne nécessitent aucun traitement spécifique parce que, par nature, elle sont déjà accessibles pour tous les threads ;
- les variables partagées, qui ont une portée locale, nécessitent une déclaration locale en tant que pointeurs, qui doivent être initialisés pour adresser les variables originales ; toutes les références aux variables originales seront remplacées par l'utilisation de ces pointeurs.

Plus en détails le transformateur OMPi utilise un analyseur lexicale écrit en flex ainsi qu'un analyseur syntaxique écrit en bison. À l'issue de l'analyse syntaxique le code source initial est représenté sous la forme d'un arbre syntaxique. Cet arbre syntaxique est accompagné d'une table de symboles, implémentée sous la forme d'une table de hachage pour améliorer la performance dans le travail avec des chaînes de caractères. Toutes les transformations du code sont opérées exclusivement au niveau de l'arbre syntaxique qui, à la dernière étape, est parcouru pour générer le code final.

Il faut noter le fait que l'arbre syntaxique n'est pas décoré : ainsi, en particulier, les identificateurs à l'intérieur d'une expression ne sont pas liés avec leurs déclarations correspondantes. Par ailleurs, la table de symbole est non-persistante ce qui implique que, pour toute analyse ou modification de l'arbre syntaxique, la table de symbole doit être recréée en parcourant l'arbre.

III.1.2 Représentation AST

Le transformateur du code OMPi s'appuie sur une structure classique d'arbre syntaxique. Lorsque le transformateur construit l'arbre syntaxique il s'appuie sur sept structures principales de nœuds : **astexpr**, **astspec**, **astdecl**, **aststmt**, **ompcon**, **ompdir** et **ompclause**, utilisées respectivement pour les expressions et les spécificateurs d'une déclaration, le déclarateur d'une déclaration, instruction, structure OpenMP, directive OpenMP ou clause OpenMP. En effet chaque nœud est un pointeur vers une structure de données associée.

La racine de l'arbre initial est un nœud instruction de type BlockList. Les nœuds sont connectés par des pointeurs descendants, depuis la racine vers les feuilles, et les parcours se font dans la même direction. Cependant certaines transformations nécessitent

un parcours inverse depuis les feuilles vers la racine. C'est pour permettre cela que les nœuds de type instructions (aststmt) disposent d'un pointeur parent qui pointe vers le sous-arbre parent ; ces liens sont initialisés au moment de la création de l'arbre syntaxique et ils sont maintenus tout au long des transformations en utilisant des fonctions spécifiques qui vérifient et recréent ces liaisons. La figure III.1 présente l'exemple d'un sous-arbre syntaxique pour l'expression

```
for (i...)
  {
    c[i] = a[i] + b[i];
  }
```

III.1.3 Les analyseurs lexical et syntaxique

Les analyseurs lexical et syntaxique se conforment à la norme C99 [ISO99] et à la version 3.0 d'OpenMP. L'analyseur lexical considère que le fichier à prendre en charge a déjà été traité par une passe du préprocesseur C. L'analyseur syntaxique s'appuie sur un table de symboles pour déterminer si un élément est une variable (identificateur) ou un nom de type ; il a aussi un double rôle dans les phases de transformation : analyser, aussi bien un fichier source qu'une chaîne de caractères.

III.1.4 Notre compilateur dérivé d'OMPi

L'objectif de la thèse a été en particulier d'étendre le compilateur OMPi pour qu'il génère du code CUDA C capable d'être exécuté sur des architecture NVIDIA. La démarche pour générer du code CUDA est un cas particulier, mais les mêmes stratégies peuvent être appliquées pour générer d'autres types de code, pour d'autres architectures. Une brève liste des modifications qui ont été opérées est donnée ci-dessous et sera détaillée dans les sections suivantes :

- les deux analyseurs, lexical et syntaxique, ont été modifiés pour prendre en charge les spécificités du langage CUDA (par exemple les qualificateurs des fonctions, ou les types des variables) ;
- la transformation du `pragma omp parallel for` produira une fonction noyau et non pas une fonction standard ;
- tous les transferts mémoire CUDA vers/depuis l'accélérateur sont rajoutés dans le code principal ainsi que le passage de paramètres pour le noyau et l'appel de celui-ci ;
- les variables privées et partagées sont transférées sur la carte graphique et, avec pour chacune un traitement spécifique correspondant à sa propre portée, par rapport à la hiérarchie mémoire sur le périphérique.

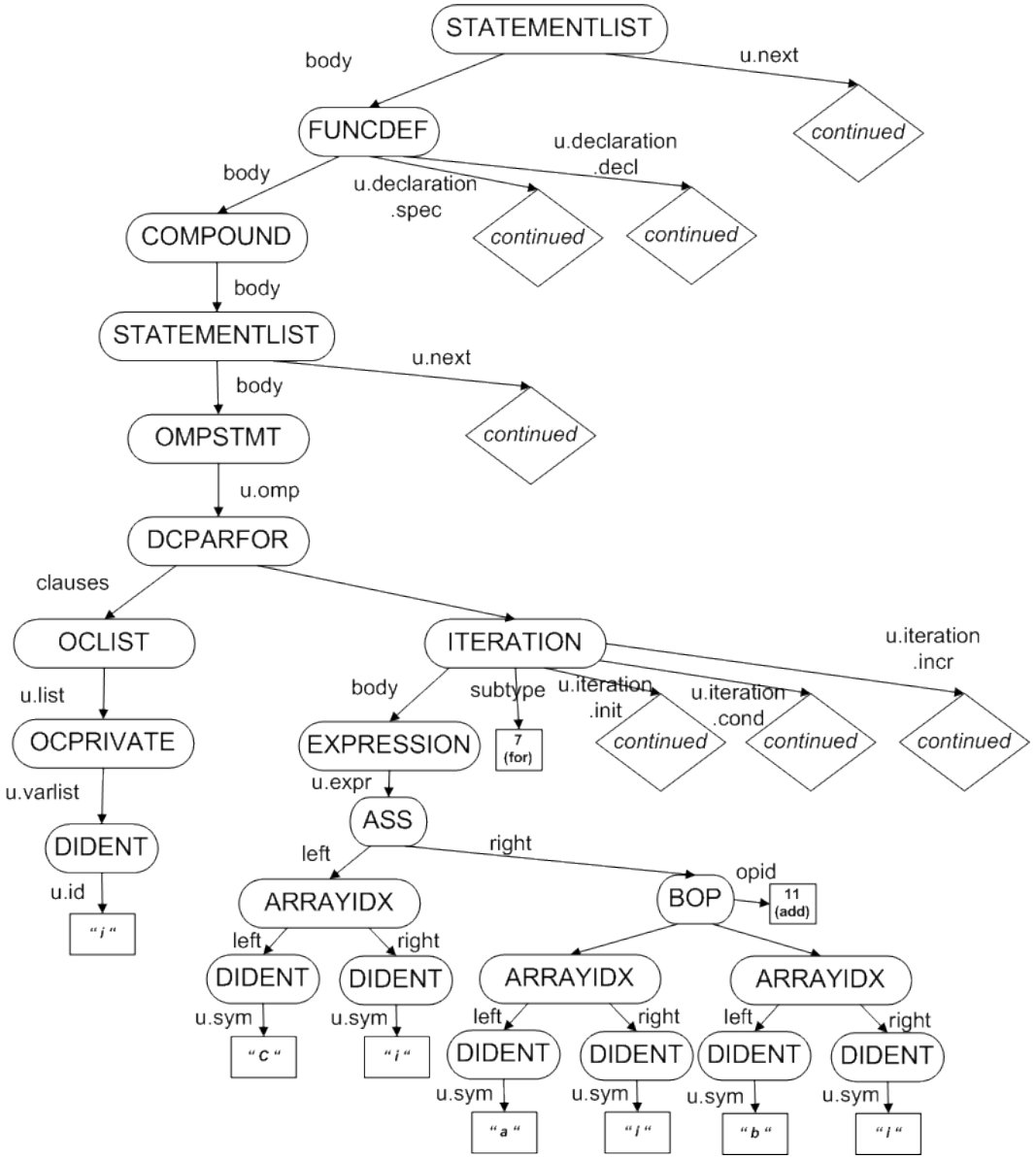


Figure III.1: Exemple d'un arbre syntaxique pour l'expression `for (i...) {c[i] = a[i] + b[i];}`

III.2 Spécificités du langage CUDA et intégration à l'analyseur syntaxique

Le modèle de programmation CUDA a été conçu pour adresser de manière transparente la scalabilité des applications et pour tirer profit de la croissance constante du nombre d'unités de calcul. Afin de maintenir une faible courbe d'apprentissage pour les programmeurs familiers avec le langage standard C, le langage C CUDA expose les fonctionnalités GPU à l'aide d'un ensemble minimal d'extensions du langage et d'une bibliothèque d'exécution. La liste complète des extensions est présentée dans l'annexe

B du Guide de programmation CUDA de NVIDIA [NVIc] ; la version actuelle de notre compilateur n'en intègre que le sous-ensemble nécessaire aux transformations désirées.

Les principales modifications apportées à l'analyseur syntaxique OMPi sont nécessaires pour la prise en charge des extensions CUDA C, notamment les qualificateurs de type pour les fonctions et pour les variables.

- Les qualificateurs CUDA pour les fonctions permettent de spécifier au compilateur comment générer la fonction considérée : `__device__` (respectivement `__host__`) pour une fonction à destination de périphérique (respectivement de l'hôte) ; `__global__` pour une fonction noyau.
- Les qualificateurs pour les variables permettent de préciser la localisation des variables utilisées sur le périphérique : il s'agit de `__device__` pour des variables dans la mémoire globale, `__constant__` pour de variables dans la mémoire constante, `__shared__` pour des variables dans la mémoire partagée.
- On dispose de variables implicites en lecture seule pour spécifier les dimensions de la grille et du bloc. De même que les indices du bloc et du thread, elles ne sont pas implémentées dans l'analyseur lexical car elles sont traitées comme des variables standard.
- De la même manière le type de vecteurs `dim3` est implicite : il n'est pas rajouté à la grammaire car on utilise, à la place, l'équivalent C défini dans le *header* CUDA.

Toutes les modifications nécessaires à la prise en charge de ces éléments de langage CUDA C ont été opérées dans les fichiers flex et bison en rajoutant les règles correspondantes.

III.3 Approche pour la transformation de code

III.3.1 Objectif et méthode générale

Le parallélisme de boucle est le type de parallélisme le plus courant et le plus simple à exploiter. OpenMP fournit le pragma `parallel for` pour préciser que une boucle doit être exécutée en parallèle. Beaucoup de programmes peuvent être parallélisés avec succès juste en appliquant la directive aux boucles appropriées. Ce style de parallélisation fine est particulièrement utile car elle peut être appliquée d'une manière incrémentale : dans certains cas des boucles peuvent représenter une parte prépondérante du travail séquentiel et être des goulots d'étranglement de performance ; elles peuvent être parallélisées facilement en ajoutant des directives et en apportant de petits changements mineurs au code source.

Parce que la parallélisation incrémentale est une technique attrayante, la directive `omp parallel for` est l'une des directives les plus importantes et le plus fréquemment utilisées. Cependant, le programmeur doit choisir avec soin les boucles à paralléliser. La version parallèle du programme doit produire les mêmes résultats que la version séquentielle ; en d'autres termes, l'exactitude du programme doit être maintenue. Les codes source pris en charge par notre transformateur sont supposés être corrects du

point de vue d'OpenMP et produire des résultats exacts après compilation par un compilateur OpenMP. Les sections suivantes détailleront les étapes pour transformer un tel code source.

III.3.2 Transformation d'une boucle pour parallèle

1) La directive `omp parallel for`

La transformation d'un `pragma omp parallel for` est réalisée conformément à la version originale d'OMPi. Ainsi une directive composée `parallel for` est remplacée par une nouvelle directive `parallel` dont le corps est une directive simple `for` qui, à son tour, reprend le corps original de la directive composée (FIGURE III.2). La raison de cette transformation réside dans le modèle de gestion des directives employé par le compilateur OMPi, dans lequel chaque directive simple est traitée par une unité de transformation spécifique individuelle.

Le code initial :

```
#pragma omp parallel for <clauses>
   <corps initial>
```

Le code transformé :

```
#pragma omp parallel <qq clauses>
   #pragma omp for <qq clauses>
      <corps initial>
```

FIGURE III.2: Transformation d'une directive composé

Un point clé dans la transformation est représenté par le partage des clauses entre les deux directives : la directive `parallel` prend en charge toutes les clauses à l'exception de `private` et `firstprivate` ; ces deux dernières sont transférées vers la directive `for`.

2) La directive `omp parallel`

La transformation de la directive `parallel` est la partie centrale du tout le processus. Le noyau CUDA est créé à partir du corps de cette structure. Comme nous le verrons, le corps de la directive `for` sera le code principal du noyau qui sera exécuté sur le périphérique, et la boucle sera mappée sur le modèle de programmation GPU, en générant autant de threads que la taille du domaine de la boucle.

Le premier pas dans la transformation consiste à créer la déclaration de la fonction pour le nouveau noyau. Conforme à la norme CUDA C, celui-ci est défini comme une fonction `__global__ void`. Parce que le code initial peut contenir plusieurs régions parallèles, et comme le nom de la fonction noyau doit être unique, ce nom est généré

comme `_kernel`N`_`, où N sera incrémenté de 1 pour chaque région parallèle. Le noyau doit déclarer toutes les variables comme paramètres, qu'elles soient `shared`, `private`, `firstprivate` et `lastprivate` ; de plus la fonction dépend de toutes les variables, à la fois issues des directives `parallel` et `for`, ainsi la liste des paramètres est générée en deux étapes pour chaque pas de la transformation.
Une fois la fonction noyau générée, il s'agit de modifier le code correspondant à la région de la directive `parallel` afin de préparer l'appel du kernel et de le réaliser. Un modèle général de transformation pour la directive `parallel` est présenté dans la FIGURE III.3 :

– Dans le code original le pragma `parallel` est remplacé par un bloc d'instructions complexe qui prend en charge le lancement du noyau (voir la section III.3.4).
– Le bloc est précédé et suivi par des allocations (respectivement libérations) de mémoire CUDA et des transferts de mémoire CUDA entre le périphérique et l'hôte pour chaque variable détectée dans les clauses (voir la section III.3.3).

Le code initial :

```
#pragma omp parallel
   <corps>
```

Le code transformé – fonction main :

```
<allocation mémoire CUDA>
<copies mémoire CUDA (hôte vers péripherique)>
<appel du kernel>
<copies mémoire CUDA (péripherique vers hôte)>
<libération des ressources mémoire>
```

Le code transformé – fonction noyau :

```
__global__ void _kernel0_(<liste des paramètres>)
{
    <initialisation des block_id et thread_id>
    <corps modifié>
}
```

FIGURE III.3: Génération du noyau

3) La directive `omp for`

Le deuxième pas de la transformation concerne le traitement de la structure `for`. Le transformateur adhère à la version 3 d'OpenMP donc il prend en charge des boucles imbriquées et la clause `collapse`. Selon la présence (ou non) de la clause `collapse`, des méthodes de transformation différentes sont utilisées. Nous envisageons ici, le cas des boucles pour parallèles sans parallélisme imbriqué : la transformation des boucles pour parallèles utilisant la clause `collapse` est abordée dans la section suivante.

Sans perdre de généralité on peut considérer la boucle pour parallèle OpenMP de la figure III.4, dans laquelle nous considérons $ub \geq lb$ et $step > 0$. Il faut noter que si la condition terminale est de la forme $i <= q$, elle sera transformée en l'expression équivalente $i < ub$, où $ub = q + 1$.

```
#pragma omp for
for (i = lb; i < ub; i += step)
   <corps de la boucle>
```

Figure III.4: Une boucle pour OpenMP (sans imbrication)

Dans cet exemple il y a une seule boucle pour et il n'y a pas la clause `collapse`. Ce type de structure sera mappé sur le modèle d'exécution CUDA dans lequel le GPU génèrera les threads automatiquement au moment de l'exécution. Chaque thread GPU exécutera indépendamment le corps de la boucle `pour` sur un cœur CUDA. C'est pour cela que la structure de la boucle `pour` est enlevée complètement. Toutes les informations de la boucle (les limites inférieure et supérieure, le pas) sont récupérées et transmises au noyau, qui les utilise pour calculer un identifiant de thread unique : en utilisant les informations de la boucle `pour` on détermine le `thread_id` pour assurer un accès correct au zones mémoire. Ce type de mappage correspond dans le milieu OpenMP, à un ordonnancement statique avec une dimension 1 pour le *chunk*. Pratiquement il n'y a pas de limitation `pour` la taille de la boucle pour car le nombre de threads GPU est assez large pour accommoder des plages de données conséquentes [2].

Finalement, le corps de la boucle `pour` est transformé, avant d'être mis dans le corps du noyau : les variables privées et partagées sont remplacées par leurs équivalents dans la mémoire GPU, les indices de la boucle sont remplacés pour considérer le *threadId* tout en restant dans les limites.

4) Gestion de la clause collapse

La version 3.0 de l'OpenMP introduit un nouvelle clause pour la directive `for` : la clause `collapse` peut être utilisée pour spécifier le nombre de boucles imbriquées qui sont associées au pragma `for` (Figure III.5). Le paramètre de la clause `collapse`, qui spécifie le nombre de boucles imbriquées à paralléliser, doit être une expression entière constante positive. Par défaut si aucune clause `collapse` n'est présente, la boucle associée à la construction est celle qui suit immédiatement la directive `for`. Si plus d'une boucle est associée à la directive `for` par la clause `collapse`, l'espace d'itérations de l'ensemble des boucles est concaténé dans un seul espace d'itération linéaire, qui sera mappé sur le modèle CUDA (Figure III.6). Ainsi un seul threadId

2. Environ 2.9E+17 threads peuvent être générés depuis une grille à trois dimensions, où chaque dimension a une limite de 65535 blocs (pour une version récente d'architecture, 2.x) et un bloc peut aussi avoir un maximum de 1024 threads.

gère l'ensemble de l'espace d'itérations. Cette fois-ci les variables compteur du chaque boucle restent inchangées, mais leur valeur est recalculée chaque fois par rapport au threadId (Figure III.7).

```
#pragma omp parallel for collapse(2)
  for (i=0; i < n; i++)
    for (j=0; j < p; j++)
```

Figure III.5: Exemple de deux boucles pour imbriquées avec la clause collapse

```
int iters_i_;
int iters_j_;

iters_i_ = (n - (0)) / 1;
if (((n - (0)) / 1) % 2 != 0)
++iters_i_;
niters_ = niters_ * iters_i_;
iters_j_ = (p - (0)) / 1;
if (((p - (0)) / 1) % 2 != 0)
++iters_j_;
niters_ = niters_ * iters_j_;
```

Figure III.6: Calcul de l'espace linéaire d'itération avant l'appel du noyau (cas de deux boucles imbriquées)

```
int iter_ = thread_id;
pp_ = 1;
j = (0) + 1 * ((iter_ / pp_) % iters_j_);
pp_ *= iters_j_;
i = (0) + 1 * ((iter_ / pp_) % iters_i_);
```

Figure III.7: Calcul des identifiants de la boucle dans le noyau (cas où deux boucles imbriquées ont été linéarisées)

Notons toutefois que nous travaillons selon l'hypothèse que le code à transformer est un code OpenMP valide et correct, donc le transformateur ne fait aucune vérification de non dépendance pour les variables des boucles à paralléliser.

III.3.3 La visibilité des variables (privées / partagées)

Les cartes graphiques exposent une hiérarchie mémoire à plusieurs niveaux.

Plusieurs threads au sein d'un programme OpenMP parallèle s'exécutent à l'intérieur du même espace d'adressage partagé et ils peuvent partager l'accès à des variables au

sein de cet espace d'adressage commun. Le partage des variables entre les threads rend la communication entre les threads très simple : les threads envoient des données à d'autres threads en affectant des valeurs aux variables partagées et reçoivent des données en lisant les valeurs de leur côté. En effet tout ce système représente la simplicité de ce modèle de programmation parallèle, qui le rend attractif.
Le modèle permet également à une variable d'être désignée comme privée à chaque thread plutôt que partagée, afin de permettre aux threads de faire leurs calculs de manière indépendante. Chaque thread a un accès restreint à une copie privée de cette variable avec une portée interne à la zone parallèle. Certaines variables peuvent également être déclarées en tant que `firstprivate` ou `lastprivate` si leur contenu doit être initialisé pour les threads à partir d'une valeur antérieure, ou (respectivement) récupérée à partir de la dernière modification effectuée par un thread (le dernier thread à modifier sa copie de la variable).

Tous ces types de variables ont besoin d'un traitement spécifique lors de la transformation de code afin d'avoir un comportement conforme au sein du modèle de programmation CUDA. Dans tous le cas des variables équivalentes doivent être allouées sur le GPU afin que les noyaux puissent accéder aux données. Les prochaines parties sont dédiées aux transformations nécessaires pour chacun de ces types de variables.
Dans tous ces cas, une première passe de la transformation consiste à déterminer si la variable considérée est un scalaire ou un tableau car ces derniers nécessitent un traitement à part lié à la récupération de leur(s) dimension(s). La taille du tableau pour chaque dimension doit être connue au moment de la compilation [3]. C'est pour cette raison que notre transformateur prend en charge uniquement le tableaux alloués statiquement. Le premier pas dans leur transformation consiste alors à récupérer leur dimension, ainsi que leur taille dans chaque dimension, afin que les tableaux correspondants soient alloués sur la carte graphique.

1) Variables partagées

Les variables partagées OpenMP sont transférées sur la carte graphique au début et récupérées à la fin d'exécution du noyau. Celles-ci sont placées dans la mémoire globale du périphérique, afin que tous les threads puissent y accéder durant tout le temps d'exécution de noyau. Pour assurer l'unicité des noms de variables, celles qui sont déclarées sur le périphérique ont leurs noms précédés par le préfixe « `_d_` » (« d » pour *device*).

Les étapes de transformation sont les suivantes :
– La variable identifiée comme partagée est isolée et sa déclaration est récupérée.
– Le nom de la variable est isolé et un nouveau symbole est créé en le préfixant par

3. Les tableaux alloués dynamiquement ne peuvent pas être ni partagés ni privés, dans la mesure où c'est le pointeur qui est partagé ou privé (alors que les données sur le tas sont dans un espace partagé).

« _d_ ».
– Le type de la variable est récupéré et sera utilisé pour une nouvelle déclaration sur la carte graphique.

La déclaration effective sur le périphérique est différente selon le type de variable. Pour une variable scalaire les étapes de la transformation du code sont présentées par la figure III.8. La variable est d'abord allouée avec la fonction `cudaMalloc`, puis transférée avec la fonction `cudaMemcpy` en spécifiant la direction de copie avec le paramètre `cudaMemcpyHostToDevice`. Une fois l'exécution du noyau terminée la variable est transférée depuis le périphérique avec la même fonction `cudaMemcpy` mais en spécifiant la direction inverse de la copie avec le paramètre `cudaMemcpyDeviceToHost`. Tout à la fin, la ressource est libérée avec la fonction `cudaFree`.

Le code initial :

```
float sum;
<initialisations du code sequentiel>
#pragma omp parallel shared(sum)
    <boucle parallèle>
```

Le code transformé – fonction main :

```
float sum;
<initialisations du code sequentiel>
float _d_sum;
cudaMalloc(&_d_sum, 1 * sizeof(sum));
cudaMemcpy(_d_sum, sum, 1 * sizeof(sum), cudaMemcpyHostToDevice);
<appel du kernel>
cudaMemcpy(sum, _d_sum, 1 * sizeof(sum), cudaMemcpyDeviceToHost);
cudaFree(_d_sum);
```

Figure III.8: Transformation d'une variable partagée scalaire

Les variables de type tableau sont déclarées directement sur le périphérique en utilisant le qualificateur de type `__device__`. Il n'y a pas d'allocation proprement dite car la variable est allouée de manière statique au moment de l'exécution. Ce type d'allocation pour le tableau permet de garder à l'intérieur du noyau un accès dans toutes les dimensions sans linéarisation des indices. La figure III.9 présente la transformation du code pour une variable partagée de type tableau : le transfert vers la carte graphique est réalisé en utilisant la fonction `cudaMemcpyToSymbol` avec le paramètre `cudaMemcpyHostToDevice` ; une fois l'exécution du noyau terminé la variable est transférée depuis le périphérique avec la fonction `cudaMemcpyFromSymbol` avec le paramètre `cudaMemcpyDeviceToHost` ; comme il n'y pas d'allocation dynamique, il n'y a pas de libération de ressources à la fin.

Comme nous l'avons indiqué, l'ensemble des transformations est réalisé par des modifications de l'arbre syntaxique. La figure III.10 présente l'exemple du sous-arbre syn-

Le code initial :

```
double a[512][512];
<initialisations du code sequentiel>
#pragma omp parallel shared(a)
    <boucle parallèle>
```

Le code transformé – fonction main :

```
double a[512][512];
<initialisations du code sequentiel>
__device__ double _d_a[512][512];
//pas d'allocation
cudaMemcpyToSymbol(_d_a, a, 512 * 512 * sizeof(a[0][0]), 0, \\
    cudaMemcpyHostToDevice);
<appel du kernel>
cudaMemcpyFromSymbol(a, _d_a, 512 * 512 * sizeof(_d_a[0][0]), 0, \\
    cudaMemcpyDeviceToHost);
//pas de libération des ressources
```

Figure III.9: Transformation d'une variable partagée tableau (dimension 2)

taxique pour l'expression $c[i] = a[i] + b[i]$. Comme on le voit sur cet exemple chaque nœud correspondant à une case du tableau est remplacé par un autre sous-arbre, correspondant à la variable équivalente sur la carte graphique, l'index lui-même étant également transformé [4].

2) Variables privées

Une variable privée OpenMP est une variable dont chaque thread peut accéder à une copie, qui lui est propre. Afin de reproduire le comportement d'une variable privée, disponible par duplication pour chaque thread CUDA, notre traducteur alloue un tableau ayant la même taille que le nombre total de threads qui vont être utilisés pour l'exécution du noyau. L'allocation de ce tableau est réalisée dans la mémoire globale du GPU, selon une implémentation semblable à celle utilisée pour les variables partagées. En particulier, puisque ce tableau est en mémoire globale, son nom est préfixé par « `_d_` ». Les temps d'accès aux données de la mémoire pâtissent d'une grande latence. Ainsi, pour optimiser l'utilisation du tableau permettant d'implémenter une variable privée, nous utilisons la mémoire partagée des multiprocesseurs pour accélérer les accès. Étant donné que seules 32 cases du tableau sont nécessaires lors de l'exécution d'un

4. La méthode de transformation est présentée en III.3.2 : *Transformation d'une boucle pour parallèle*

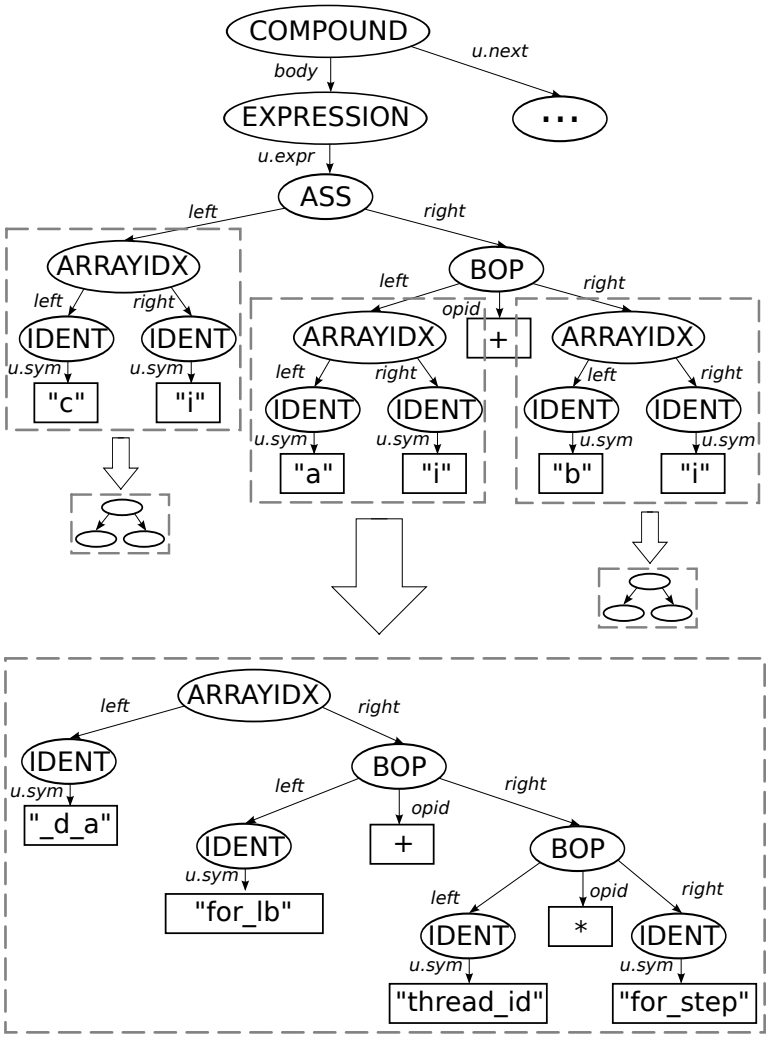


Figure III.10: Transformation de l'arbre syntaxique dans le cas de l'utilisation d'un tableau : exemple de l'expression c[i]=a[i]+b[i]

warp [5], les cases spécifiques exactes sont copiées dans cette zone spécifique partagée sur le multiprocesseur où s'exécute le warp.

La figure III.12 synthétise le schéma de transformation pour une variable privée scalaire x et la figure III.11 détaille les allocations mémoire correspondantes (pour distinguer le tableau local au multiprocesseur, le nom de la variable correspondante est préfixé par « `_d_sh_` »).

5. L'ordonnancement des threads CUDA est détaillé en II.2.1

Le code initial :

```
int j;
#pragma omp parallel private(j)
    <boucle parallèle>
```

Le code transformé – fonction main :

```
int j;
int (* _d_j);
cudaMalloc(&_d_j, CUDA_thread_num * sizeof(j));
```

Le code transformé – fonction noyau :

```
int _d_sh_j[ 32];
_d_sh_j[thread_id % 32] = _d_j[thread_id];
<calculs>
_d_j[thread_id] = _d_sh_j[thread_id % 32];
```

Figure III.11: Transformation d'une variable privée scalaire

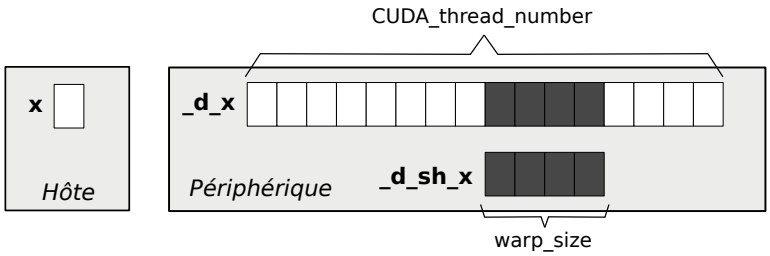


Figure III.12: Schéma de transformation pour une variable privée scalaire

3) Les variables « firstprivate » et « lastprivate »

Le même mécanisme utilisé pour l'implémentation des variables privées peut être mis en place pour les clauses `firstprivate` and `lastprivate`. La version actuelle de notre traducteur de code n'implémente pas ces deux fonctionnalités, mais pour compléter la présentation nous détaillons ici les modifications nécessaires :

– Pour les variables `firstprivate` la valeur initiale doit être répliquée dans toutes les cases du tableau dans la mémoire globale du périphérique afin qu'il soit accessible à chaque thread.
– Pour les variables `lastprivate` une des cases du tableau doit être choisie pour être transférée vers l'hôte à la fin de l'exécution du noyau. La spécification OpenMP désigne la valeur retenue comme étant la dernière valeur écrite par le dernier thread exécuté mais, du fait du modèle d'exécution CUDA il est impossible de déterminer le dernier thread qui a modifié une case du tableau (en effet les threads s'exécutent de façon synchrone au sein d'un warp). Pour cette raison, sans perte de généralité, une

case aléatoire du tableau peut être choisie comme dernière valeur et être retournée vers l'hôte.

III.3.4 L'appel d'un kernel

Le langage CUDA propose deux méthodes pour l'appel des kernels dans le mode *C runtime* (figure III.13). La première méthode, utilisée par la plupart des programmeurs, se base sur une syntaxe étendue d'un appel classique de fonction, qui permet de spécifier les paramètres de configuration pour le kernel entre des triples crochets (⋘ et ⋙). Cette méthode bien que simple, ne respecte pas le standard C99. Afin de supporter les appels de kernels dans notre transformateur de code, nous nous sommes basés sur la deuxième méthode offerte par le langage CUDA, mais moins utilisée à cause de sa complexité.
Dans cette approche alternative trois appels de fonction sont utilisés dans une ordre bien précis pour initialiser les paramètres de kernel (voir figure III.13) :

Première méthode (la plus utilisée ; ne respecte pas le standard C99) :

```
_kernelN_ <<< grid_size, threads_per_block >>> (A, B);
```

Deuxième méthode (moins utilisée ; respecte le standard C99) :

```
cudaConfigureCall(grid_size, threads_per_block, 0, NULL);

offset = 0;
ptr = (void*)(size_t)A;
ALIGN_UP(offset, __alignof(ptr));
cudaSetupArgument(&ptr, sizeof(ptr), offset);

offset += sizeof(ptr);
ptr = (void*)(size_t)B;
ALIGN_UP(offset, __alignof(ptr));
cudaSetupArgument(&ptr, sizeof(ptr), offset);

cudaLaunch("_kernelN_");
```

Figure III.13: Appel d'un kernel : les deux méthodes

- La fonction `cudaConfigureCall` permet de spécifier les tailles de la grille et du bloc, comme dans un appel classique utilisant des chevrons.
- Chaque paramètre du kernel est déposé dans une structure de type pile en utilisant la fonction `cudaConfigureCall`.
- Enfin la fonction `cudaLaunch` exécutera le kernel dont le nom est spécifié en paramètre. Cette dernière étape conclut l'appel du kernel après avoir récupéré les para-

mètres envoyés dans la pile.

Grace à cette deuxième méthode, implémentée dans notre compilateur, la grammaire nécessaire pour supporter les appels de kernels reste inchangée : en effet cela évite de devoir y ajouter la reconnaissance des appels de fonction pour lesquels les paramètres sont spécifiés entre triple chevrons.

Remarque : Dans une démarche similaire on peut simplifier la déclaration de la taille des blocs (`blockDim`) et des grilles (`gridDim`). Dans l'approche CUDA classique ces déclarations utilisent un constructeur de type C++ pour des variables de type `dim3`. Par contre la bibliothèque CUDA permet de gérer ce type sous la forme d'une structure à trois champs. Ainsi une déclaration avec initialisation du type
`dim3 grid_size(num_blocks_x, num_blocks_y);` est équivalente à deux affectations standards : `grid_size.x = num_blocks_x; grid_size.y = num_blocks_y;`.

III.4 Analyse expérimentale

III.4.1 Structure de l'outil

1) Conditions expérimentales

Nous avons suivi l'évolution du compilateur OMPi au plus près. Au début de nos travaux nous avons utilisé la version 1.0, qui supportait la version 2.5 d'OpenMP. Avec sa version 1.2 le compilateur a apporté le support pour OpenMP 3.0. Un important travail de réintégration de notre surcouche a été mené pour pouvoir profiter des nouvelles fonctionnalités, et notamment le support pour la clause `collapse`.

Pour la compatibilité avec les différents environnements de programmation CUDA nous avons testé chaque version dès son apparition. Alors nous avons testé nos codes générés avec les version 2, 3 et 4 du CUDA.

Pour mener des tests de transformation nous avons utilisé la version 4.5 de GCC ainsi que la version 4.0 de nvcc.

2) Transformation en trois passes

Comme présenté dans la figure III.14 le processus de transformation comporte trois étapes indépendantes : une étape de *préprocessing*, la transformation principale, et une étape de nettoyage final du code.

La première étape de la transformation, correspondant à la commande ci-dessus, consiste en effet en une étape de préprocessing relativement classique. A cette étape on utilise un préprocesseur C avec support pour les directives OpenMP (gcc 4.2 ou plus

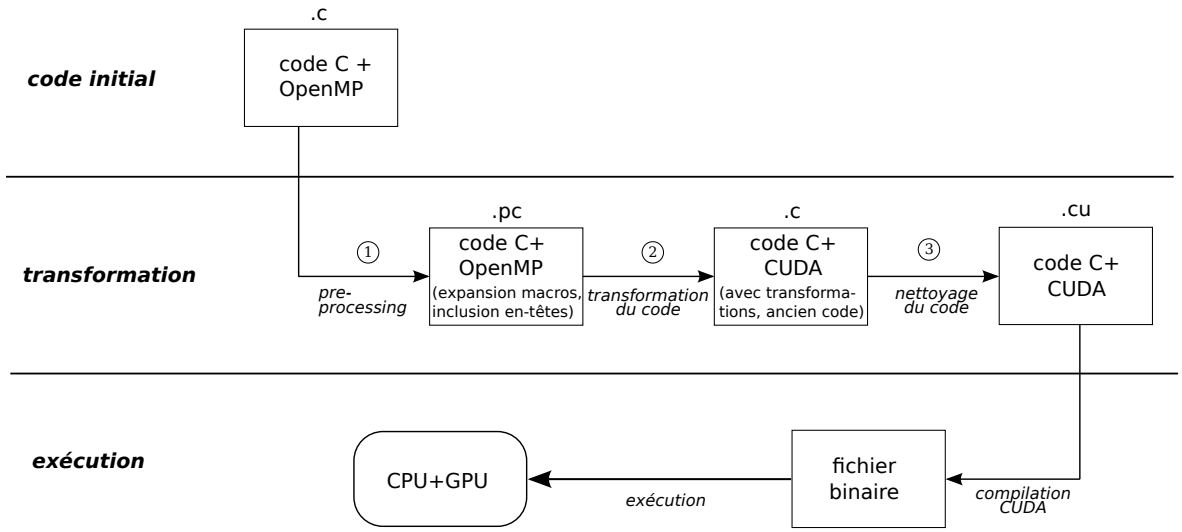


Figure III.14: Schéma de transformation

récent dans notre cas). On génère un fichier d'extension `.pc` qui intègre le code des fichiers d'entête (*header*) ainsi que les expansions des macros du code original. Il est important d'inclure le chemin du compilateur OMPi pour permettre au compilateur d'utiliser une version modifiée du fichier « omp.h » à la place du fichier standard GCC : en effet la complexité du fichier standard le rend difficile à analyser par OMPi.

```
gcc -std=gnu99 -E -U__GNUC__ -D_OPENMP=200805 -D_REENTRANT \
   -I${OMPI_PATH}/runtime ${FILE}.c -o ${FILE}.pc
```

La deuxième étape réalise la transformation effective du code, comme elle est décrite dans la section précédente. Une fois le code complètement transformé en modifiant l'AST contenu dans la mémoire vive, l'arbre est « affiché » en utilisant des fonctions spécifiques qui effectuent un parcours en profondeur afin de générer le code transformé. Le résultat est stocké dans un fichier `.c`.

```
${OMPI_PATH}/compiler/ompi/ompi ${FILE}.pc __ompi__ > ${FILE}.c
```

La troisième étape est nécessaire pour effectuer un nettoyage final du code CUDA afin d'éliminer toutes les traces résiduelles d'OpenMP. Il faut aussi déclarer les kernels CUDA comme `extern "C"` afin d'éliminer les problèmes de compilation avec le compilateur NVIDIA. Le script de nettoyage, dont le code est présenté par la figure III.15, permet de générer un nouveau fichier résultat, d'extension `.cu`. Ce nouveau fichier peut ensuite être compilé avec le compilateur NVIDIA nvcc, selon le fichier de Makefile présenté par la figure III.16, pour obtenir le fichier binaire exécutable sur les architectures compatibles CUDA.
Cette étape pourrait ne pas être nécessaire dans les prochaines versions du compilateur dans la mesure où, en travaillant avec des versions figées d'OMPi, OpenMP et CUDA on n'aurait pas à préserver des éléments pouvant servir ultérieurement.

```
#!/bin/bash
FILE=$1
TMP_FILE=tmp.c

#Add extern C for kernel functions
sed 's/__global__ void _kernel/extern "C" __global__ \
  void _kernel/g' $FILE > $TMP_FILE

#Remove the omp runtime references
head -n 2 $TMP_FILE > ${TMP_FILE}u
sed '2d' $TMP_FILE | grep -E 'omp_|omp.h' -v >> ${TMP_FILE}u
mv ${TMP_FILE}u ${FILE%.*}.cu
rm $TMP_FILE
```

FIGURE III.15: Le script de nettoyage : script_clear_ompi_header.sh

```
CC=nvcc
CUDA_DIR= /opt/cuda/4.0
CUDA_ARG= -arch=sm_21 -m 64 --ptxas-options=-v -g -G -keep -v
SDK_DIR= ${CUDA_DIR}/sdk
CFLAGS= -I. -I${CUDA_DIR}/include -I${SDK_DIR}/C/common/inc
LDFLAGS= -L${CUDA_DIR}/lib64 -L${SDK_DIR}/lib \
 -L${SDK_DIR}/C/common/lib
LIB= -lm -lrt
SOURCES= vector_cuda_ompi.cu
EXECNAME= vector_cuda_ompi

all:
   $(CC) $(CUDA_ARG) -o $(EXECNAME) $(SOURCES) $(LIB) \
         $(LDFLAGS) $(CFLAGS)

clean:
   $(CC) $(CUDA_ARG) -clean -o $(EXECNAME) $(SOURCES) $(LIB) \
         $(LDFLAGS) $(CFLAGS)
   rm -f *.o core
```

FIGURE III.16: Le fichier Makefile pour la compilation CUDA du fichier transformé

III.4.2 Résultats

Notre présentons dans cette section un exemple complet de transformation, correspondant à l'algorithme classique de multiplication de deux matrices. Bien que simple, ce problème est la base d'une série d'algorithmes de méthodes numériques utilisés dans

plusieurs domaines allant de la simulation chimique à l'analyse financière. Nous utilisons cet algorithme pour illustrer nos résultats au long de ce manuscrit.

1) Codes en extensions

La figure III.1 présente la version OpenMP du produit matriciel pour deux matrices carrées, allouées statiquement. Pour la parallélisation des trois boucles imbriquées on a naturellement utilisé le pragma `omp for`, et les variables des matrices `a`, `b`, `c` ont été déclarées comme partagées. Pour tirer profit du fait que les deux premières boucles sont indépendantes nous avons rajouté la clause `collapse(2)` permettant de paralléliser les deux niveaux de boucles (*nested parallelism* : parallélisme imbriqué).

Listing III.1: Produit matriciel - version OpenMP

```
#include <omp.h>

#define NRA 16384   /* lignes de la matrice A */
#define NCA 16384   /* colonnes de la matrice A */
#define NCB 16384   /* colonnes de la matrice B */

int main (int argc, char *argv[]) {
  int   tid, nthreads, i, j, k;
  double a[NRA][NCA];
  double b[NCA][NCB];
  double c[NRA][NCB];

/* Initialisation des matrices */
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        a[i][j]= (double) (i+j);
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        b[i][j]= (double) (i*j);
    for (i=0; i<NRA; i++)
      for (j=0; j<NCB; j++)
        c[i][j]= 0.0;

#pragma omp for collapse(2) shared(a,b,c) private(k)
    for (i=0; i<NRA; i++) {
       for(j=0; j<NCB; j++) {
         for (k=0; k<NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
         }
      }
    }
/*** Fin du région parallèle***/
}
```

La figure III.2 présent quant a elle la version du code transformé par notre outil.

Listing III.2: Produit matriciel - version CUDA, après nettoyage du code

```
int __original_main(int argc, char * (* argv))
{
  int i, j, k;
  double a[ 1024][ 1024], b[ 1024][ 1024], c[ 1024][ 1024];

  for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
      a[i][j] = (double) (i + j);
  for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
      b[i][j] = (double) (i * j);
  for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
      c[i][j] = 0.0;

    int niters_ = 1;
    int CUDA_thread_num = 0;

    /* Calcul des variables de la boucles */
    int iters_i_;
    int iters_j_;

    iters_i_ = (1024 - (0)) / 1;
    if (((1024 - (0)) / 1) % 2 != 0)
      ++iters_i_;
    niters_ = niters_ * iters_i_;
    iters_j_ = (1024 - (0)) / 1;
    if (((1024 - (0)) / 1) % 2 != 0)
      ++iters_j_;
    niters_ = niters_ * iters_j_;
    CUDA_thread_num = niters_;
    int threads_per_block = 256;
    int num_blocks = (int) ((float) (CUDA_thread_num +   \\
        threads_per_block - 1) / (float) threads_per_block);
    int max_blocks_per_dimension = 65535;
    int num_blocks_y = (int) ((float) (num_blocks +      \\
        max_blocks_per_dimension - 1) /                  \\
        (float) max_blocks_per_dimension);
    int num_blocks_x = (int) ((float) (num_blocks +      \\
        num_blocks_y - 1) / (float) num_blocks_y);
    dim3 grid_size;

    grid_size.x = num_blocks_x;
    grid_size.y = num_blocks_y;
    grid_size.z = 1;
    // Aucune allocation dynamique pour les tableaux
    cudaMemcpyToSymbol(_d_c, c, 1024 * 1024 *     \\
        sizeof(c[0][0]), 0, cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(_d_a, a, 1024 * 1024 *             \\
        sizeof(a[0][0]), 0, cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(_d_b, b, 1024 * 1024 *             \\
        sizeof(b[0][0]), 0, cudaMemcpyHostToDevice);
```

```
53
54      cudaConfigureCall(grid_size, threads_per_block);
55      int offset = 0;
56      void * ptr;
57
58      ptr = (void *) iters_i_;
59      ALIGN_UP(offset, __alignof(iters_i_));
60      cudaSetupArgument(&ptr, sizeof((iters_i_)), offset);
61      offset += sizeof((iters_i_));
62      ptr = (void *) iters_j_;
63      ALIGN_UP(offset, __alignof(iters_j_));
64      cudaSetupArgument(&ptr, sizeof((iters_j_)), offset);
65      offset += sizeof((iters_j_));
66      cudaLaunch(_kernel0_);
67      cudaMemcpyFromSymbol(c, _d_c, 1024 * 1024 *       \\
68          sizeof(c[0][0]), 0, cudaMemcpyDeviceToHost);
69      cudaMemcpyFromSymbol(a, _d_a, 1024 * 1024 *           \\
70          sizeof(a[0][0]), 0, cudaMemcpyDeviceToHost);
71      cudaMemcpyFromSymbol(b, _d_b, 1024 * 1024 *           \\
72          sizeof(b[0][0]), 0, cudaMemcpyDeviceToHost);
73    }
74  }
75
76
77  __global__ void _kernel0_(int iters_i_, int iters_j_)
78
79  int block_id = blockIdx.x + gridDim.x * blockIdx.y;
80  int thread_id = blockDim.x * block_id + threadIdx.x;
81
82  /* (l32) #pragma omp parallel shared(a,b,c) -- body moved below */
83  {
84    /* #pragma omp for collapse(2) private(i,j,k) */
85    int j;
86    int i;
87    int k;
88    int pp_ = 1;
89
90        {
91          int iter_ = thread_id;
92
93          pp_ = 1;
94          j = (0) + 1 * ((iter_ / pp_) % iters_j_);
95          pp_ *= iters_j_;
96          i = (0) + 1 * ((iter_ / pp_) % iters_i_);
97          {
98            for (k = 0; k < 1024; k++)
99              {
100               _d_c[i][j] += _d_a[i][k] * _d_b[k][j];
101             }
102         }
103 }
104
105 /* OMPi-generated main() */
```

```
106 int main(int argc, char **argv)
107 {
108   int _xval = 0;
109   _xval = (int) __original_main(argc, argv);
110   return (_xval);
111 }
```

Ces deux fichiers sont toutefois difficile a comparer globalement, et nous présentons dont ci-après une comparaison synthétique, en deux temps, afin de caractériser les transformations réalisées.

2) Comparaison synthétique

Pour permettre de comparer effectivement les codes initial et final, nous en menons ici une comparaison synthétique.

La première comparaison concerne la structure des fichiers en présence. La partie déclaration globale des matrices ainsi que l'initialisation reste inchangées entre les deux codes (III.1 : lignes 9-22 et III.2 : lignes 4-14), alors que la fonction `main` (III.1 : ligne 7) devient `__original_main` (III.2 : ligne 1). La nouvelle fonction main dans le code transformé contient tout simplement un appel de la fonction `main` initiale. Il faut aussi noter le fait que les macros pour les tailles des matrices ont été inclues dans le code dans l'étape de pre-processing.

La principale modification est celle des boucles pour parallèles. La transformation principale concerne donc la ligne 24 qui contient le pragma composé `parallel for`, ainsi que les clauses pour les visibilités des variables (Figure III.17). Dans un premier temps ce pragma est transformé en interne en deux pragmas indépendants, qui ultérieurement sont marqués en commentaire dans le code final (III.2 : lignes 83 et 85). Après cette transformation il y a une série de transformations spécifiques pour chacune d'entre elles.

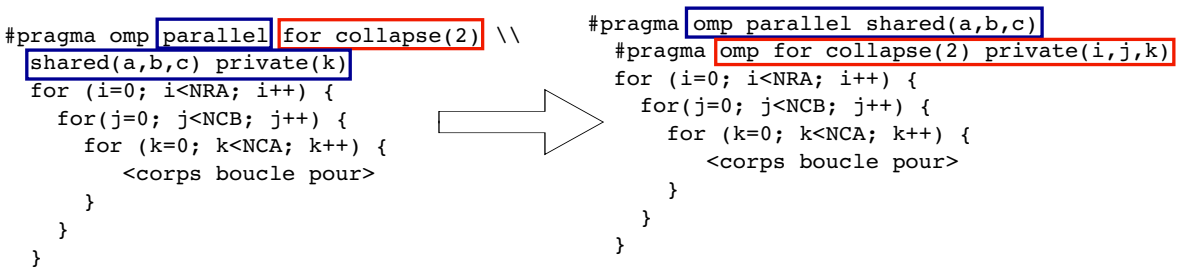


Figure III.17: Transformation du pragma composé

– D'abord le pragma `omp parallel` a été remplacé par un appel de kernel (III.2 : ligne 67, Figure III.18 cadre vert). L'appel lui-même est précédé par la construction de la pile de transfert des paramètres (III.2 : lignes 55-66), la construction de la pile étant elle aussi précédée par la configuration de la taille des grilles et des blocs pour l'appel

Figure III.18: Transformation du pragma parallel

de kernel (III.2 : lignes 31-41), basé sur des éléments constituants les boucles pour. Notez que le cadre vert est l'équivalent d'un appel de kernel plus courant qui utilise les trois chevrons (voir III.3.4 : *L'appel d'un kernel*). Toutes les variables partagées se voient déclarer des copies sur le périphérique : ces copies reçoivent les valeurs initiales *via* des transferts mémoire avant l'appel du kernel (III.2 : lignes 47-52) et sont recopiées à la fin d'exécution dans les variables hôtes (III.2 : lignes 67-72). Dans notre cas toutes les variables partagées sont des tableaux ce qui explique la présence des `cudaMemcpyToSymbol` et `cudaMemcpyFromSymbol` (Figure III.18 cadre violet), autrement les variables partagées doivent être alloués dynamiquement et ensuite transférées (voir 1) : Variables partagées).

– La transformation implique aussi de générer une fonction kernel qui sera exécutée sur le GPU. A l'intérieur du kernel nous pouvons noter le fait que les deux boucles extérieures ont été déroulées en une seule, elle même éliminée et remplacée par le calcul d'un `thread_id`, qui fera office d'indice de boucle Figure III.18 cadre rouge).
– Le pragma `omp for` ainsi que son corps sont transférés à l'intérieur de kernel (Fi-

Figure III.19: Transformation du pragma for

gure III.19 cadre vert). Le nombre de boucles parallèles imbriquées, indiqué par la clause `collapse`, est remplacé par un bloc d'instructions prenant en charge le calcul des variables d'itération équivalentes mais basé sur le `thread_id` (III.2 : lignes 91-96). De ce fait toutes les références à des variables d'itération restent inchangées. Par contre toutes les références à des copies sur le périphérique sont remplacées dans le code original. Les informations des boucles sont aussi utilisées pour calculer l'espace d'itération et ainsi évaluer le nombre total de threads nécessaire pour exécuter la problème (Figure III.19 cadre violet).

Ces commentaires concernent le code initial, comparé au code généré par notre compilateur. A titre de comparaison, nous avons également confronté le code généré avec notre outil avec celui écrit à la main directement en langage CUDA. Bien évidement les codes obtenus sont en tout point semblables, ou équivalents lorsque les transformations nous ont obligés à utiliser des schémas CUDA moins classiques, mais d'effet identique (c'est le cas des appels de kernel par exemple). Concernant les exécutables, nous avons

bien sûr vérifié que les traitements réalisés sont identiques et sont réalisés dans des temps absolument identiques.

III.5 Amélioration du code transformé

III.5.1 Les leviers d'optimisations

Pour tirer un maximum de performance des cartes graphiques, chaque multiprocesseur a la capacité de traiter activement plusieurs blocs en même temps (voir le chapitre I.3.1). Le nombre total effectif dépend du nombre de registres utilisés par chaque thread à l'intérieur d'un bloc, ainsi que de la quantité de mémoire partagée requise par bloc. Les kernels doivent avoir des besoins de ressources minimaux pour mieux occuper chaque multiprocesseur car les registres et la mémoire partagée du multiprocesseur sont répartis entre tous les threads des blocs actifs [6].
Par ailleurs, les caractéristiques matérielles et logicielles imposent des limitations lorsqu'on programme avec CUDA (voir tableau III.1).

Spécifications techniques	Compute Capability						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Nombre maximal de dimensions d'une grille	2				3		
Dimension maximale sur x pour une grille	65535					2^{31}-1	
Dimension maximale sur y et z pour une grille	65535						
Nombre maximal de dimensions d'un bloc	3						
Dimension maximale sur x et y pour un bloc	512				1024		
Dimension maximale sur z pour un bloc	64						
Nombre maximal de threads / bloc	512				1024		
Nombre maximal de blocs résidents / SMX	8					16	
Nombre maximal de warps résidents / SMX	24		32		48	64	
Nombre maximal de threads résidents / SMX	768		1024		1563	2048	
Nombre de registres (32 octets) / SMX	8 K		16 K		32 K	64 K	
Nombre maximal de registres / thread	128				63		255
Taille maximale de la mémoire partagée / SMX	16 Ko				48 Ko		

TABLE III.1: Limite matériel du CUDA

NVIDIA propose un calculateur d'occupation livré avec ses outils de développement [CUD]. Celui-ci se présente sous la forme d'une feuille de calcul et permet d'explorer les compromis entre les nombres de threads et de blocs actifs, par rapport au nombre de registres et à la quantité de mémoire partagée. Trouver la bonne combinaison peut

6. Sont considérés comme actifs à un moment donné les blocs qui sont traités par un multiprocesseur.

augmenter considérablement les performances des appels de kernel. Cependant, s'il n'y a pas suffisamment de registres ou de la mémoire partagée disponible par multiprocesseur pour traiter au moins un bloc, le kernel ne pourra pas démarrer. L'interface de la feuille de calcul (Figure III.20) est divisée en plusieurs zones : certaines permettent de saisir des informations textuelles sur les caractéristiques de l'exécution a évaluer ; les autres sont des zones graphiques permettant de rendre compte de l'évaluation de l'exécution, selon différents métriques. Ainsi, en spécifiant la *compute capability* et les caractéristiques de l'exécution que sont les nombres de threads/bloc et de registres/-thread ainsi que la taille de la mémoire partagée (en octets) par bloc, le calculateur nous permet d'obtenir une évaluation selon quatre métriques :
– nombre de threads actifs / multiprocesseur ;
– nombre de warps actifs / multiprocesseur ;
– nombre de blocs actifs / multiprocesseur ;
– occupation de chaque multiprocesseur, qui est le rapport entre le nombre de warps actifs et le nombre maximal de warps actifs envisageable par multiprocesseur (voir les limites en fonction de la *compute capability* dans le tableau III.1).

Un *tuning* fin des paramètres d'exécution peut apporter une plus-value à l'application qui se traduit par des accélérations supérieures lorsque l'occupation est optimisée. Il y a donc lieu de fixer les paramètres d'exécution au mieux (nombre de thread/bloc, nombre de registres/thread, quantité de mémoire partagée/bloc) en respectant les limites liées à la *compute capability*, afin de maximiser l'occupation des multiprocesseurs (dernière métrique). On peut obtenir une assez bonne estimation des « bons » paramètres d'exécution en les récupérant automatiquement à l'issue d'une étape de précompilation avec le compilateur nvcc, en utilisant l'option `-ptxas-options=-v`. Sur la base de ces valeurs, on peut ensuite essayer de les faire varier pour améliorer l'occupation attendue. Dans la section suivante nous présentons une possibilité de réglage automatique de ces paramètres qui peut être intégrée dans notre outil de transformation.

III.5.2 *Tuning* automatique des paramètres d'exécution

Un appel de kernel CUDA est entièrement déterminé par le nombre de blocs dans une grille et le nombre de threads dans un bloc. Nous avons vu précédemment que leur choix a une importance capitale pour la performance de l'application.

Dans notre transformateur ces éléments sont calculés d'une manière statique de la façon suivante :
– on calcule le nombre total de threads nécessaire pour couvrir l'espace d'itérations linéarisé ;
– on fixe le nombre maximal de threads par bloc, manuellement, en fonction du *compute capability* ;
– on détermine le nombre total des blocs nécessaire (en tenant compte de la limite antérieure), ainsi que le nombre de grilles nécessaire pour accommoder tous ces threads.

Une fois calculées, ces valeurs sont utilisées pour initialiser le kernel ; mais elles peuvent être sous-optimales. Pour cette raison nous envisageons d'optimiser notre transforma-

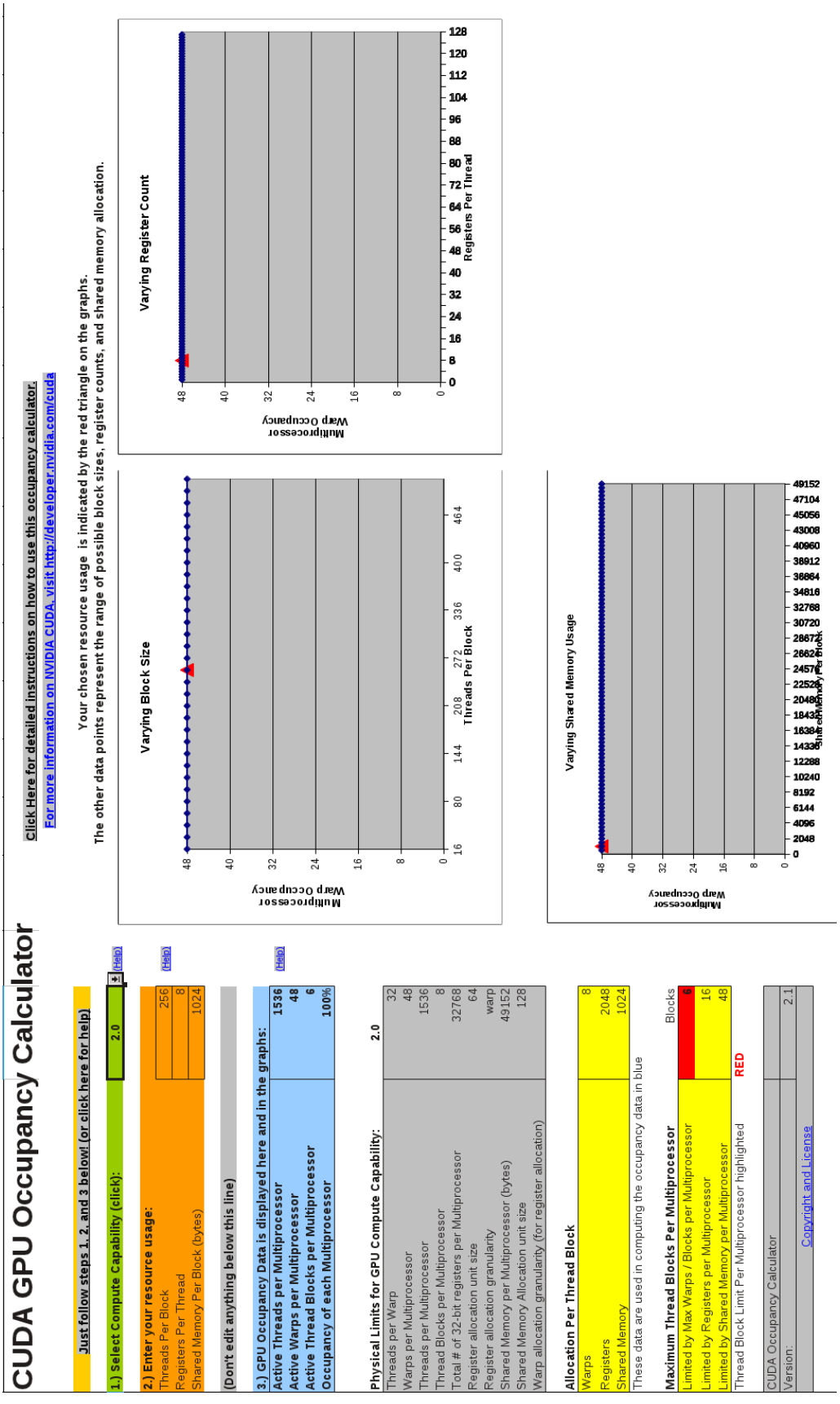


FIGURE III.20: Calculateur d'occupation CUDA

teur pour avoir une meilleure occupation des multiprocesseurs. Ce fonctionnement sera possible en utilisant une compilation en deux temps :
- une étape de pré-compilation (en utilisant les options du compilateur nvcc) produira un code intermédiaire avec des paramètres par défaut
- le code sera sondé afin de récupérer les paramètres utiles, ainsi que les sorties du compilateur nvcc concernant l'utilisation des registres et mémoire partagée
- ces informations seront utilisées en entrée d'un script qui génèrera les mêmes calculs que le calculateur d'occupation CUDA ;
- l'étape finale sera l'incorporation de ces paramètres, et leur utilisation en amont de la seconde étape de compilation permettra d'obtenir alors des codes bien dimensionnés pour le support envisagé.

Cette approche a été déjà testée pour créer un outil d'auto-tunning [Sør12], avec de bons résultats. Les niveaux 1 et 2 des bibliothèques CUBLAS sont optimisés avec cette technique.

Néanmoins cette méthode n'est pas une solution universelle, car l'occupation n'est pas le seul point à optimiser. En principe, cette piste doit effectivement être envisagée dans le cas où le kernel est limité par la bande passante où la bande passante utilisée est loin de la bande passante crête. Toutefois certaines études précisent en effet que, lorsque l'utilisation de la bande passante est presque maximale, l'augmentation de l'occupation des multiprocesseurs risque de ne rien apporter : dans ce cas il est pertinent de reconsidérer le programme dans sa conception, en visant d'exploiter plus efficacement le parallélisme d'instructions [Vol10].

III.6 Conclusions

Le développement présenté dans ce chapitre consiste en un transformateur de code permettant, à partir d'un programme source écrit en C+OpenMP, de générer un programme équivalent en C+CUDA permettant d'adresser les GPU de NVIDIA. Le choix d'OpenMP est naturel car il permet d'exprimer le parallélisme d'une manière simple et efficace, notamment le parallélisme de boucle. Celui de retenir CUDA comme langage cible est motivé par le fait qu'il s'impose désormais comme un standard pour le GPU Computing sur les cartes NVIDIA (les plus répandues dans le domaine du HPC) ; cependant nous aurions pu, avec la même démarche, décider de générer du code pour tout autre type de carte graphique ou d'accélérateur.

Au chapitre 2 nous avons vu qu'il existe différents outils, développés en même temps que le nôtre, dont l'objectif est de générer du code spécifique pour accélérateurs à partir de code de haut niveau exprimant le parallélisme à l'aide de directives (voir OpenMPC). De plus le nouveau langage OpenACC, fortement poussé par le groupe NVIDIA-PGI-CAPS, reprend la même idée d'utiliser des directives pour exhiber la performance des accélérateurs, avec la promesse de préserver la facilité de programmation. Ce constat nous permet de valider la piste retenue pour notre travail.

Dans le cadre de ce chapitre nous avons vu que l'outil proposé est fonctionnel et répond bien aux desiderata initiaux : être facile à utiliser et générer du code lisible par un humain. Pour mener à bien nos travaux nous avons choisi de nous appuyer sur le transformateur de code OMPi, qui supporte maintenant la norme OpenMP v3.0, et nous l'avons modifié pour pouvoir générer du code CUDA. Notre outil est assez évolutif et la généralité des transformations permet d'envisager un autre langage cible pour le code final généré, pour supporter par exemple le langage OpenCL. En effet l'utilisation de l'arbre syntaxique comme forme intermédiaire de représentation a l'avantage de permettre une déconnexion du langage proprement dit, ce qui dégage la généralité de l'approche.

Notre outil supporte un sous-ensemble des directives OpenMP, incluant notamment les boucles parallèles imbriquées. Ce type de construction peut être transformé naturellement en un kernel qu'il est possible d'exécuter sur une carte graphique. Pour ce faire la transformation consiste en plusieurs étapes successives, mais les choix faits au moment de la transformation sont en lien étroit avec l'architecture matérielle cible. Plusieurs optimisations du code généré ont été proposées et implémentées, notamment l'utilisation optimale de la hiérarchie mémoire des cartes graphiques (utilisation de la mémoire partagée pour réduire les coûts d'accès à la mémoire globale). D'autres optimisations, comme le *tuning* des paramètres d'exécution des kernels, sont présentées formellement pour des évolutions ultérieures.
Pour conclure ce chapitre nous avons mené un exposé détaillé de la transformation d'un code spécifique, pour présenter en parallèle le code initial et la version générée par notre outil.

Nous avons pu constater dans l'évolution des machines parallèles durant ces dernières années, que de plus en plus de supercalculateurs du TOP500 disposent d'accélérateurs, notamment des cartes graphiques. A condition d'en faciliter l'utilisation, nous pensons qu'elles ouvrent le chemin vers l'exascale. Pour cette raison une démarche logique est de s'intéresser non seulement à améliorer la programmation d'une carte graphique seule au sein d'un nœud, mais aussi de penser au passage à l'échelle avec plusieurs accélérateurs au sein d'un nœud, voire plusieurs nœuds multi-cartes. Dans cette logique, le chapitre suivant présente l'étude que nous avons menée pour proposer un modèle de gestion des nœuds de calcul multiGPU permettant de tirer profit de toute leur puissance de calcul.

IV

Multi-GPU

Sommaire

Le calcul distribué ou réparti, ou encore partagé, est l'action de répartir un calcul ou un traitement sur plusieurs microprocesseurs et plus généralement plusieurs unités de calcul. Le calcul est souvent distribué sur des clusters de calcul spécialisés, mais peut aussi être réalisé sur des stations informatiques individuelles à plusieurs cœurs. De ce fait nous distinguons plusieurs niveaux de parallélisme : entre les nœuds et au sein d'un même nœud. La configuration classique consiste à avoir des serveurs reliés entre eux et à l'intérieur de chaque nœud plusieurs processeurs multicœurs. Les deux niveaux de parallélisme sont orthogonaux, c'est à dire que chacun est géré indépendamment. En rajoutant dans cette configuration des accélérateurs nous apportons une troisième dimension au parallélisme. Cette configuration consiste en effet en une grappe de multiGPU (Figure IV.1). Au niveau des communications entre les nœuds le modèle de programmation est sans doute basé sur le passage de message (MPI) à cause des caractéristiques des systèmes à mémoire physiquement distribuée. Par contre à l'intérieur du nœud il y a plusieurs possibilités pour gérer plusieurs cartes accélératrices, notamment le passage de messages ou programmation en mémoire partagée.

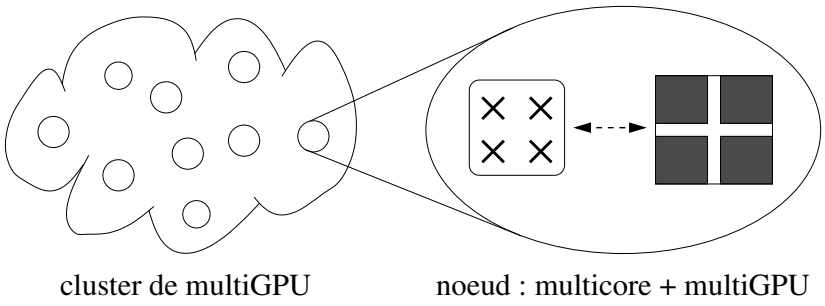


Figure IV.1: Grappe de multiGPU

Dans ce chapitre nous présentons une étude comparative des approches pour gérer un nœud multiGPU. Les détails de l'implémentation sont présentés en utilisant le langage de programmation CUDA, mais sans perdre de généralité dans le contexte de la programmation des accélérateurs. Cette étude a donné lieu a une publication, présentée dans le cadre de la conférence « IEEE International Conference on Intelligent Computer Communication and Processing » : ICCP2010 à Bucarest, Roumanie [NKJ10].

IV.1 La problématique : gérer un nœud de calcul multiGPU

Dans les applications multiGPU, les codes de calcul intensif sont déportés sur les GPU. Les accélérateurs sont des elémentes discrets et qui ne peuvent pas communiquer directement entre eux. Alors les CPU sont en charge de les piloter, en gérant particulièrement la sélection correcte des GPU et les transferts mémoire. Comme indiqué précédemment nous nous concentrons sur cette gestion à l'intérieur d'un seul nœud qui peut supporter une ou plusieurs cartes graphiques discrètes, voire une ou plusieurs cartes multiGPU (cartes Tesla/Fermi dans notre cas).

Le contrôle des contextes GPU et leur alimentation en données peut reposer sur des processus lourds (MPI) ou des threads (processus légers : OpenMP/pThreads), mais un thread/processus ne peut faire exécuter du code GPU que sur un seul périphérique à un instant donné. Ainsi pour partager le travail sur un nombre donné de GPU on utilise autant de threads/processus CPU, chacun avec son propre contexte GPU associé.

En réalité les applications multiGPU consistent en des applications classiques multi-threadées pour lesquelles les portions de code de calcul intensif sont chargées sur le GPU avec les données à traiter. Par ailleurs, toutes les API [ou bibliothèques] multi-threadées ou avec passage de messages peuvent être utilisées pour développer de telles applications, car la gestion des threads CPU et la communication inter-processus sont complètement orthogonales par rapport aux API permettant d'externaliser les calculs sur accélérateur (API CUDA dans notre cas). Le premier point clé est cependant la sélection correcte des GPU, parmi ceux disponibles, qui repose sur des éléments liés aux threads/processus eux-mêmes (voir la section IV.2.3).
Un hôte peut disposer d'une ou plusieurs cartes graphiques, soit une ou plusieurs cartes Tesla/Fermi dans notre étude. L'API CUDA fournit des fonctions pour énumérer les périphériques, interroger leurs propriétés et sélectionner le plus approprié pour les exécutions du kernel. Plusieurs threads de l'hôte peuvent faire exécuter du code sur le même GPU mais, par essence même de la fonctionnalité de sélection des cartes GPU, un thread hôte ne peut externaliser du code que sur un unique périphérique à un moment donné. Ainsi, pour partager le travail sur p GPU en même temps, un programme a besoin de p threads hôte, chacun avec son propre contexte. De même, une ressource CUDA associée à l'exécution d'un thread hôte ne peut pas être utilisée par un autre thread hôte.
Par ailleurs, les threads en charge du contrôle des contextes GPU peuvent correspondre soit à des processus lourds (MPI, ...) soit à des processus légers (OpenMP, pthreads).

IV.2 Contrôle des GPU avec OpenMP ou MPI

IV.2.1 Description du problème cible

Notre étude porte sur les différences entre les deux approches de la programmation multiGPU proposées, selon plusieurs perspectives. Afin d'en présenter les détails, nous avons développé un algorithme parallèle de multiplication de deux matrices carrées A et B (voir Figure IV.2). Bien que simple, ce problème est la base d'une série d'algorithmes de méthodes numériques utilisés dans plusieurs domaines allant de la simulation chimique à l'analyse financière. Il s'agit d'un problème standard de calcul intensif, avec une complexité de $O(n^3)$ dans sa forme séquentielle pour deux matrices $n \times n$, ce qui permet une analyse pertinente des temps d'exécution.

Pour répartir le calcul, les matrices sont décomposées en différents blocs, qui sont alors déployés sur les différentes unités de calcul. Le calcul à proprement parler, effectué sur les GPU, utilise la librairie NVIDIA CUBLAS. Nous pouvons noter que cela impose de

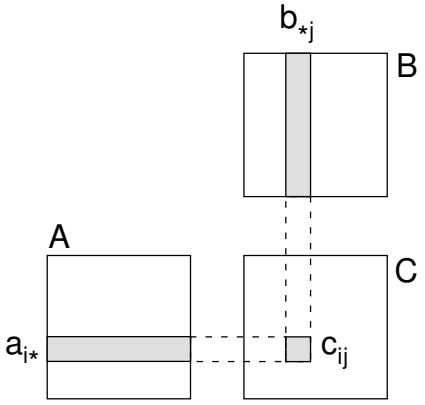


FIGURE IV.2: Produit matriciel : c_{ij} est le produit scalaire des vecteurs a_{i*} et b_{*j}

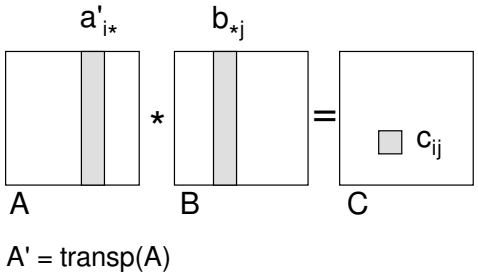


FIGURE IV.3: Produit matriciel (CUBLAS impose le stockage des matrices en colonnes)

stocker toutes les matrices/sous-matrices en colonnes [NVIa], et par conséquent d'en transposer certaines (voir Figure IV.3). Afin d'assurer une meilleure clarté de l'exposé, ces détails spécifiques ne seront pas décrits explicitement dans la suite.

Comme indiqué au début de ce chapitre, notre objectif est de proposer une mise en œuvre de calculs au sein d'un nœud multiGPU, en composant des approches basées sur la programmation en mémoire partagée ou par échange de messages. Nous avons programmé les implémentations en langage C, respectivement avec OpenMP et avec MPI.
Nous donnons ici les principaux points qui différencient les deux approches ; d'autres détails d'implémentation, concernant notamment les schémas d'allocation mémoire, sont précisés en IV.3.2 : *Détails d'implémentation*, au sein de la partie concernant l'étude expérimentale.

IV.2.2 Partitionnement des données

Sans même considérer que ce sont des GPU qui vont prendre en charge les calculs effectivement répartis, les unités de calcul doivent recevoir des portions des matrices A et B. Nous avons choisi de partitionner la matrice A et d'en répartir les blocs entre les différents threads/processus. Pour ce qui concerne le partitionnement des données, la

différence entre les deux implémentations réside dans la décomposition éventuelle de la matrice B :
– selon un modèle à mémoire partagée, les threads disposent d'une quantité plus importante de mémoire, en commun, et la matrice B peut ne pas être partitionnée (Figure IV.4a)
– utiliser un modèle à mémoire distribuée impose d'expliciter les communications : la matrice B doit donc être fractionnée (Figure IV.5a) et ses sous-matrices transmises entre les processus (nous avons retenu une topologie logique de communication en anneau [Qui03]). Techniquement cela impose uniquement d'utiliser un procédé de *buffering* afin que chaque processus dispose de deux emplacements mémoire pour y stocker les deux versions de la sous-matrice de B (versions actuelle et future).

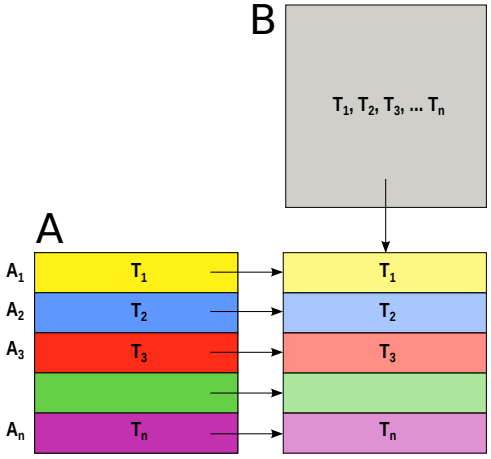
Afin de rendre les deux implémentations équivalentes, il faut éviter tout surcoût lié aux communications pour l'approche distribuée. Nous avons donc mis en place un procédé de recouvrement (*overlapping*) des communications MPI par les calculs, en utilisant des primitives de communication MPI non blocantes [Mes05] : la réception de la prochaine version de la sous-matrice de B a lieu pendant les calculs sur la version actuelle.

IV.2.3 Gestion des contextes CUDA

Un kernel s'exécute sur un GPU au sein d'un contexte, créé par un CPU et attaché à ce GPU en utilisant des routines de l'API CUDA. On peut par exemple sélectionner le périphérique auquel le thread hôte va être attaché en appelant la fonction `cudaSetDevice()` de l'environnement d'exécution CUDA [NVIb]. Quand on utilise un unique GPU, le contexte est créé par défaut sur le périphérique 0. Une méthode directe pour contrôler les différents GPU consiste à utiliser autant de threads ou de processus que de GPU. On peut ainsi choisir l'index du GPU en se basant sur l'identifiant du thread OpenMP ou sur le rang du processus MPI. Il faut noter que lorsque les processus MPI sont lancés sur une grappe de nœuds, chacun contenant plusieurs cartes graphiques, le rang MPI ne suffit pas, car il dépend de l'implémentation MPI.

IV.2.4 Les deux implémentations

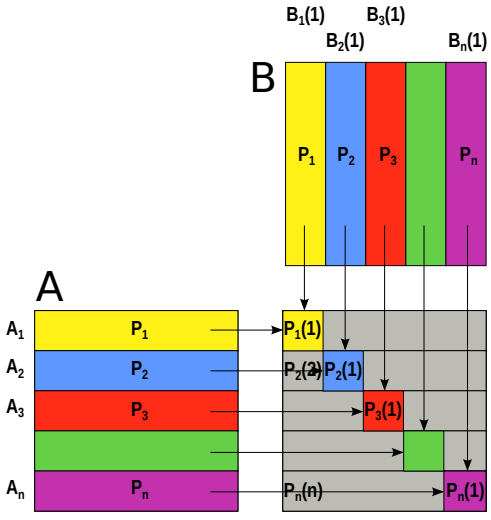
L'approche OpenMP (Figure IV.4b) utilise une région parallèle à l'intérieur de laquelle chaque thread est attaché à un GPU puis, après les différentes allocations mémoire, chacun lance sa part du calcul (pour ce faire il faut préalablement charger sur le GPU les données nécessaires et, après coup, rapatrier le résultat sur l'hôte).
Pour l'implémentation MPI (Figure IV.5b), une boucle est en charge de la répartition des tâches (après que chaque processus créé ait été lié à un GPU et ait effectué les initialisations nécessaires) :
– les sous-matrices de B sont *cyclées* au sein de la topologie MPI (alors que chaque processus conserve sa propre sous-matrice de A)
– le calcul en cours est effectué sur le GPU (avec les transferts de données correspondants entre CPU et GPU).




```
#pragma omp parallel
{   // selection du GPU
        cudaSetDevice(...)
    // allocations memoire
        # pragma omp single
            hostAlloc(A,B,C)
        cublasAlloc(dA,dB,dC)
        #pragma omp barrier
    // transferts CPU-GPU et calcul GPU
        copyH2D(Ai,dA)
        copyH2D(B,dB)
        cublasSgemm(dA,dB,dC)
        copyD2H(dC,Ci)
} // fin omp parallel
```

(a) Partition des matrices (b) Pseudo-code

FIGURE IV.4: Approche OpenMP : la matrice B est gardée en totalité en mémoire partagée




```
    MPI_Init
// les processus executent ce qui suit
// traitements préalables
    cudaSetDevice(...)
    MatrixAlloc(Ai,Bi,Ci)
    cublasAlloc(dA,dB,dC)
    MPI_Barrier
// chargement de la ligne voulue (de A)
    copyH2D(Ai,dA)
// traiter toutes les taches
    pour i allant de 1 à nb_process
    {   // circulation des blocs de B
        MPI_Irecv(Bi[Future])
        MPI_Issend(Bi[Current])
        // transferts et calcul
        copyH2D(Bi[Current],dB)
        cublasSgemm(dA,dB,dC)
        copyD2H(dC,Ci)
    }
    MPI_Barrier
```

(a) Partition des matrices (b) Pseudo-code

FIGURE IV.5: Approche MPI

Une comparaison des deux codes dans leur ensemble montre que l'implémentation OpenMP est plus facile à programmer, nécessitant moins de synchronisations explicites et de transferts de données.

IV.3 Analyse expérimentale

IV.3.1 Conditions expérimentales

Nous avons mené nos expériences sur un serveur Tesla au sein du « Centre Régional de Calcul ROMEO » de l'Université de Reims Champagne-Ardenne. La configuration, livrée par Bull, est basée sur un serveur NovaScale R425 avec deux processeurs Intel Xeon quad-core et 8 Go de mémoire vive, couplés à une NVIDIA Tesla S1070-400 contenant 4 cartes C1060 connectées en deux paires à deux ports PCI-express vers le serveur. Le système d'exploitation du serveur est *Red Hat Enterprise Linux Server*, version 5.1 (Tikanga), et nous disposons du pilote vidéo NVIDIA version 195.36.15. Les programmes ont été compilés en utilisant le compilateur nvcc (CUDA 3.0), gcc (version 4.1.2) et OpenMPI (version 1.2.3) avec l'option d'optimisation -O3.

Plusieurs tests on été menés, en utilisant 1,2 ou 4 threads/processus pour contrôler autant de cartes graphiques. Nous avons aussi mené des tests en utilisant 8 threads/-processus pour analyser le comportement dans le cas où 2 threads/processus accèdent en même temps à la même ressource (carte graphique). Nous verrons que les tailles maximales envisageables pour les matrices en entrée sont différentes selon l'implémentation et le type d'allocation mémoire. Ces détails seront présentés plus tard, à la fois parmi les détails d'implémentation (section suivante) et avec l'analyse du comportement (au sein du IV.3.3 : *Résultats et analyse*).

IV.3.2 Détails d'implémentation

1) Mesures de temps

Nous avons mesuré les temps de chaque partie du code, côté CPU et GPU, en utilisant des *timers* adaptés à nos besoins.
Le temps total de l'application a été mesuré sur l'hôte (côté CPU) en utilisant de *timers* spécifiques : la fonction `MPI_Wtime()` pour l'implémentation MPI et la fonction POSIX `gettimeofday()` pour l'implémentation OpenMP.
Du coté GPU nous avons utilisé des événements CUDA pour chronométrer les appels des fonctions pour les différents éléments caractéristiques de l'application. En effet, dans le *NVIDIA Best Practices Guide* [NVIb], les événements sont présentés comme le seul moyen de chronométrer des fonctions GPU d'un façon indépendante du système d'exploitation. L'utilisation consiste à placer deux événements dans le stream 0, synchroniser le deuxième événement pour assurer la fin des appels asynchrones (par exemple `cudaMemcpyAsync` ou les appels des kernels) et calculer la différence entre les

deux timers (Figure IV.6).

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
kernel<<<grid,threads>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time,start,stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Figure IV.6: Exemple d'utilisations des timers CUDA

2) Types d'allocation mémoire

Les transferts mémoire entre l'hôte et le périphérique doivent être effectués explicitement : la mémoire est allouée sur le CPU et sur le GPU (fonctions `malloc()` et `cudaMalloc()`), et on dispose d'instructions de transfert « *host-to-device* » et « *device-to-host* » pour transmettre les données : en fait, pour le produit $C = A \times B$, les éléments de A et B sont transférés vers les GPU et ceux de C récupérés à partir des GPU.

L'allocation standard est réalisée sur le CPU (fonction POSIX `malloc`).
Nous avons également testé les possibilités de la « *pinned memory* », littéralement *mémoire fixée* (mémoire hôte non paginable, allouée en utilisant la fonction CUDA `cudaHostAlloc`) : ce type de mémoire est alloué pour le processeur en mode « *page-locked* », ce qui garantit que le système d'exploitation ne la déplacera jamais sur le disque et qu'elle résidera toujours dans la mémoire physique (sans *swap* des pages mémoire). Cette propriété permet de donner au GPU un accès direct via le DMA (*Direct Memory Access*) pour copier ses données depuis ou vers l'hôte. Les transferts DMA ne nécessitent pas l'intervention du CPU : cette méthode permet d'obtenir des vitesses de transfert plus élevées, la seule limite étant la vitesse de transfert sur le bus PCI ou sur le bus de l'hôte.

Afin de mesurer l'impact des deux types d'allocation mémoire sur le GPU nous avons testé les deux applications en utilisant soit l'allocation standard, soit l'allocation en mémoire fixée (fonction CUDA `cudaHostAlloc`).

IV.3.3 Résultats et analyse

Pour comparer les deux stratégies de gestion des nœuds multiGPU nous avons mené nos expériences avec 1, 2 ou 4 threads (OpenMP) ou processus (MPI), contrôlant autant de GPU. Par ailleurs, pour mesurer l'incidence de ces choix dans un contexte de surcharge, nous avons également effectué des tests en utilisant 8 threads ou processus faisant réaliser des calculs sur 4 GPU, afin d'observer le comportement des programmes quand 2 threads/processus sont liés à une même carte.
Nous avons mené nos tests jusqu'aux limites constatées en pratique pour la taille des matrices d'entrée, et nous verrons que ces limites varient selon l'implémentation (OpenMP ; MPI) et avec le nombre de threads/processus et le type d'allocation mémoire envisagé.

1) Mesures réalisées

Dans tous les cas, et pour toutes les tailles de matrices, nous avons mesuré non seulement le temps global d'exécution de l'application, mais également les temps d'exécution du kernel et des copies mémoire (pour ces dernières, nous conservons le temps moyen pour un GPU, obtenu sur la base des temps cumulés et ramenés à une carte).
Les graphiques des figures IV.7, IV.8, IV.9, et IV.10 présentés ci-après, mettent en évidence les temps obtenus en fonction de la taille de la matrice considérée, pour 1 à 8 threads OpenMP (les deux premières) ou processus MPI (les deux dernières) ; ils permettent de comparer l'allocation mémoire standard (`malloc` : IV.7 et IV.8) avec l'utilisation de mémoire fixée (`HostAlloc` : IV.9 et IV.10). Ces illustrations présentent à la fois les temps d'exécution de l'application dans son ensemble (lignes continues) et ceux d'exécution du kernel (lignes brisées) et de la copie mémoire (lignes pointillées). Les temps d'exécution du kernel et les temps de copie mémoire sont des temps moyens par processus léger ou processus lourd, calculés comme la moyenne de tous les temps correspondants pour l'ensemble des threads/processus.

2) Comportement général

De façon générale nous pouvons observer que les deux implémentations passent à l'échelle dans de bonnes conditions lorsque la taille des matrices augmente.
En partant des résultats obtenus avec un seul GPU, on obtient des accélérations satisfaisantes en utilisant 2 ou 4 threads/processus pilotant autant de GPU [1].
Par contre, comme prévu, les expérimentations menées avec 8 threads/processus gé-

1. Exemples (matrice 16384x16384) :
   – MPI + malloc, 2 GPU - 2 processus, : accélération de 1,51
   – OpenMP+cudaHostAlloc, 4 GPU - 4 threads : accélération de 2,4

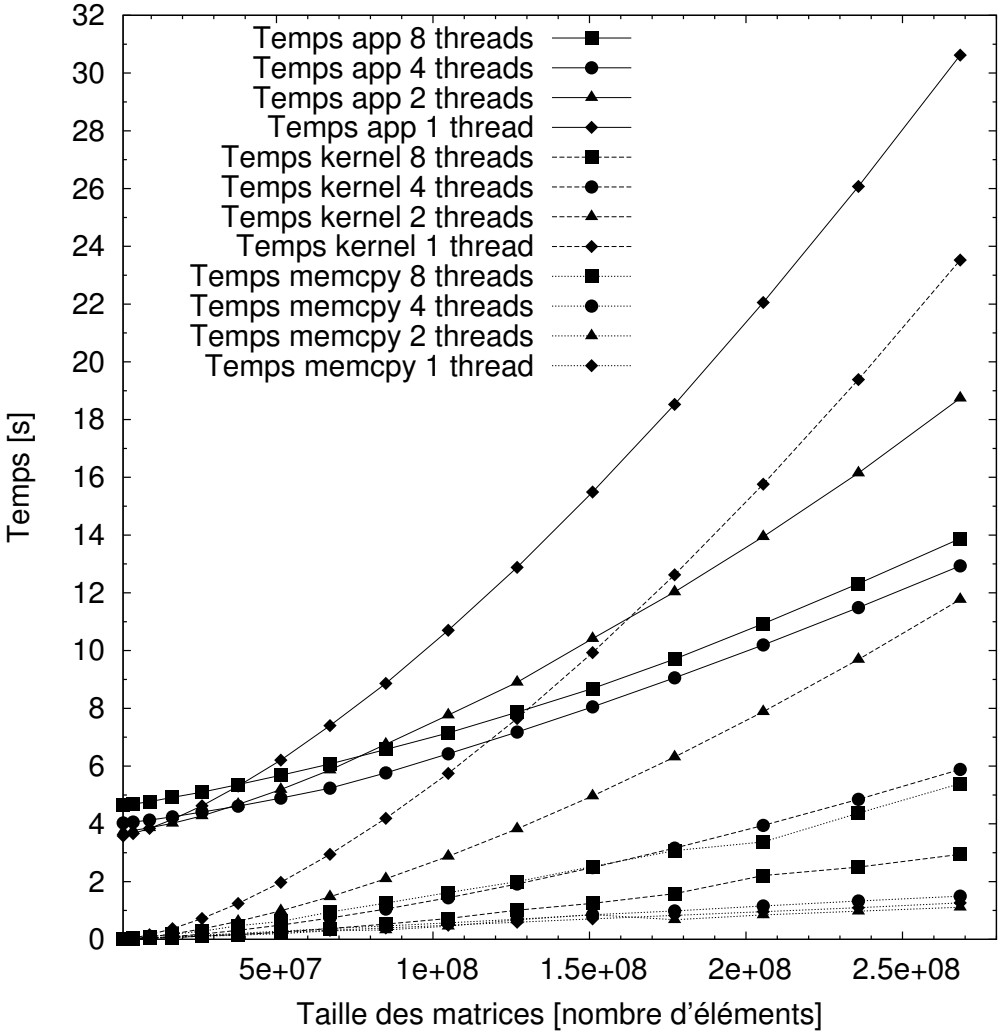


Figure IV.7: Temps d'exécution pour OpenMP avec `Malloc` (allocation CPU standard)

rant 8 contextes sur 4 GPU donnent des résultats médiocres du fait de la situation de concurrence occasionnée. En particulier la partie d'initialisation préalable au lancement des kernels occasionne déjà un surcoût de l'ordre d'une seconde, qui est d'ailleurs encore plus important dans l'approche MPI du fait de l'initialisation du milieu de communication MPI et de la gestion des processus lourds [2].

2. Exemple (matrice 16384x16384) :
OpenMP+cudaHostAlloc, 8 contextes sur 4 GPU :
- temps total de 16,14 s
- initialisation de l'ordre de 3,5 s

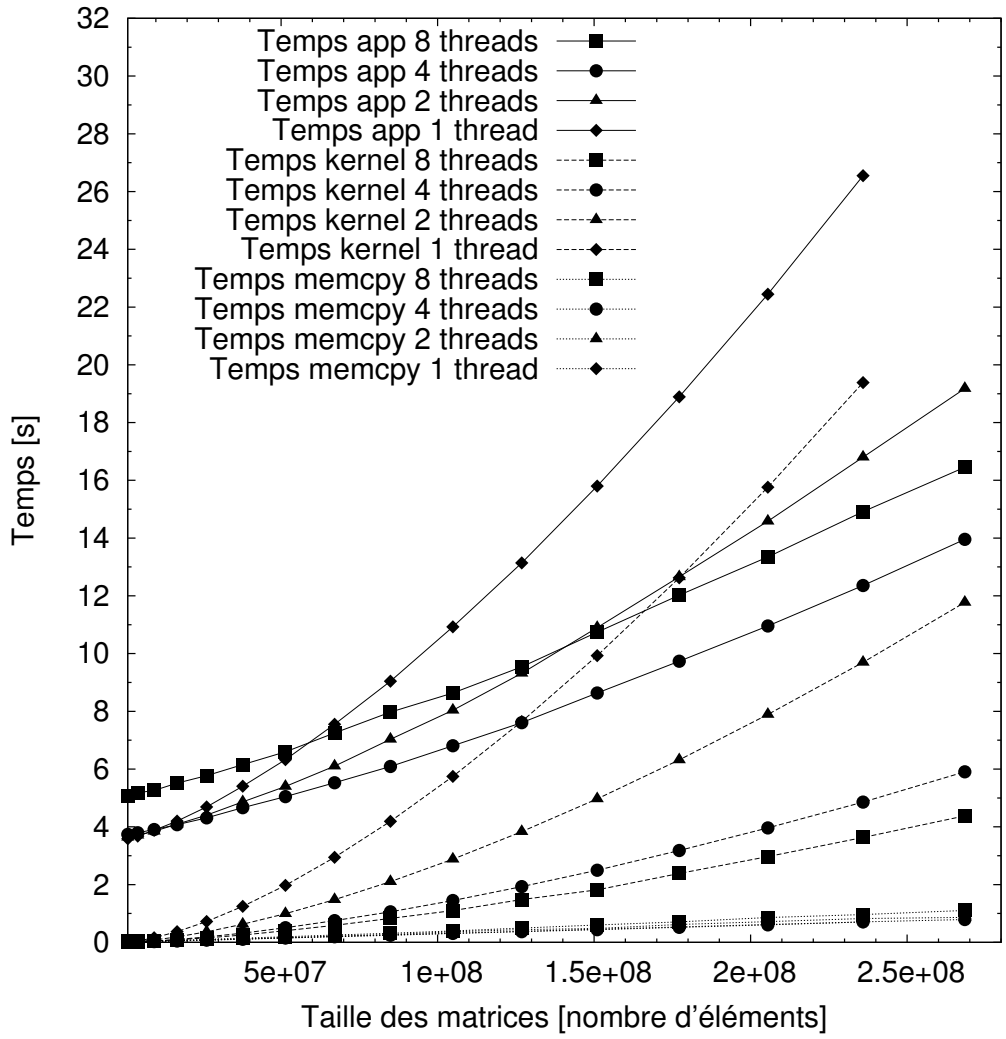


Figure IV.8: Temps d'exécution pour MPI avec Malloc (allocation CPU standard)

3) Les temps kernel

La figure IV.7 met en évidence une efficacité proche de 100% pour les temps kernel lorsqu'on passe de 1 à 2 puis 4 threads OpenMP, liée au fait que la taille des sous-matrices à traiter est systématiquement divisée par deux. On observe le même phénomène sur la figure IV.8, obtenue lorsque les cartes GPU sont gérées en utilisant des processus MPI.
Cependant, du fait du surcoût occasionné par la mise en place des threads OpenMP ou des processus MPI, nous verrons que le temps total de l'application ne suit pas une évolution tout à fait aussi favorable (voir la partie 6) : *Limites des applications et accélérations*).

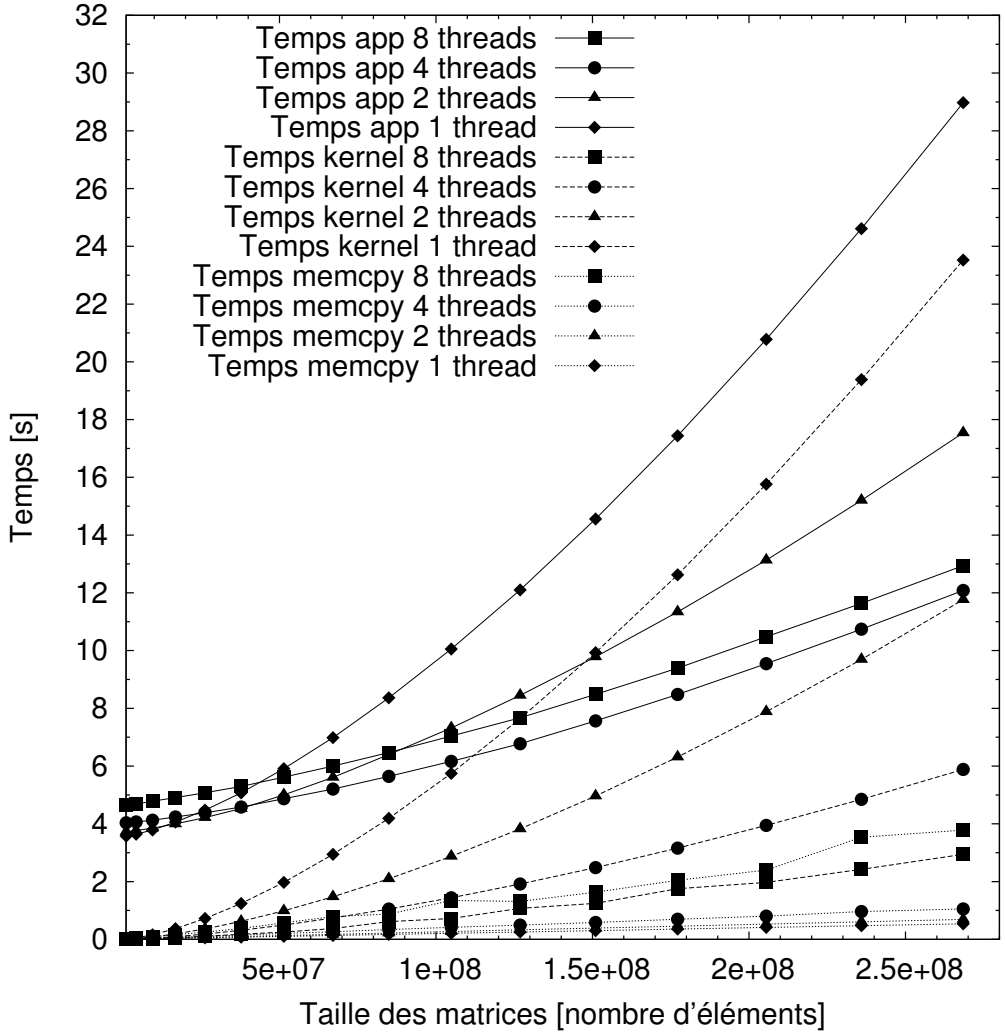


Figure IV.9: Temps d'exécution pour OpenMP avec cudaHostAlloc (mémoire fixée)

4) Les types d'allocation mémoire

Les figures IV.9 et IV.10 reprennent les mêmes courbes que celles présentées dans les figures IV.7 et IV.8, mais en illustrant l'utilisation de « *pinned memory* ». En effet ce type de mémoire, appelé aussi *mémoire figée*, permet d'obtenir des temps de transfert de données plus courts. Pour la figure IV.11 nous avons isolé les temps correspondant aux transferts mémoire, en vue de comparer une allocation classique (`Malloc`) et avec une allocation en mémoire figée (`cudaHostAlloc`). Nous pouvons facilement constater l'amélioration des temps liée à l'utilisation de mémoire *figée*, mais aussi le fait que le temps de communication est plus linéaire lorsque la taille du problème augmente. Ce comportement est bien sûr lié au fait que cette mémoire, « *page-locked* », réside à la même adresse tout au long de l'exécution, sans être transférée sur disque. Cependant la quantité de mémoire figée utilisable pour l'ensemble des GPU d'un nœud

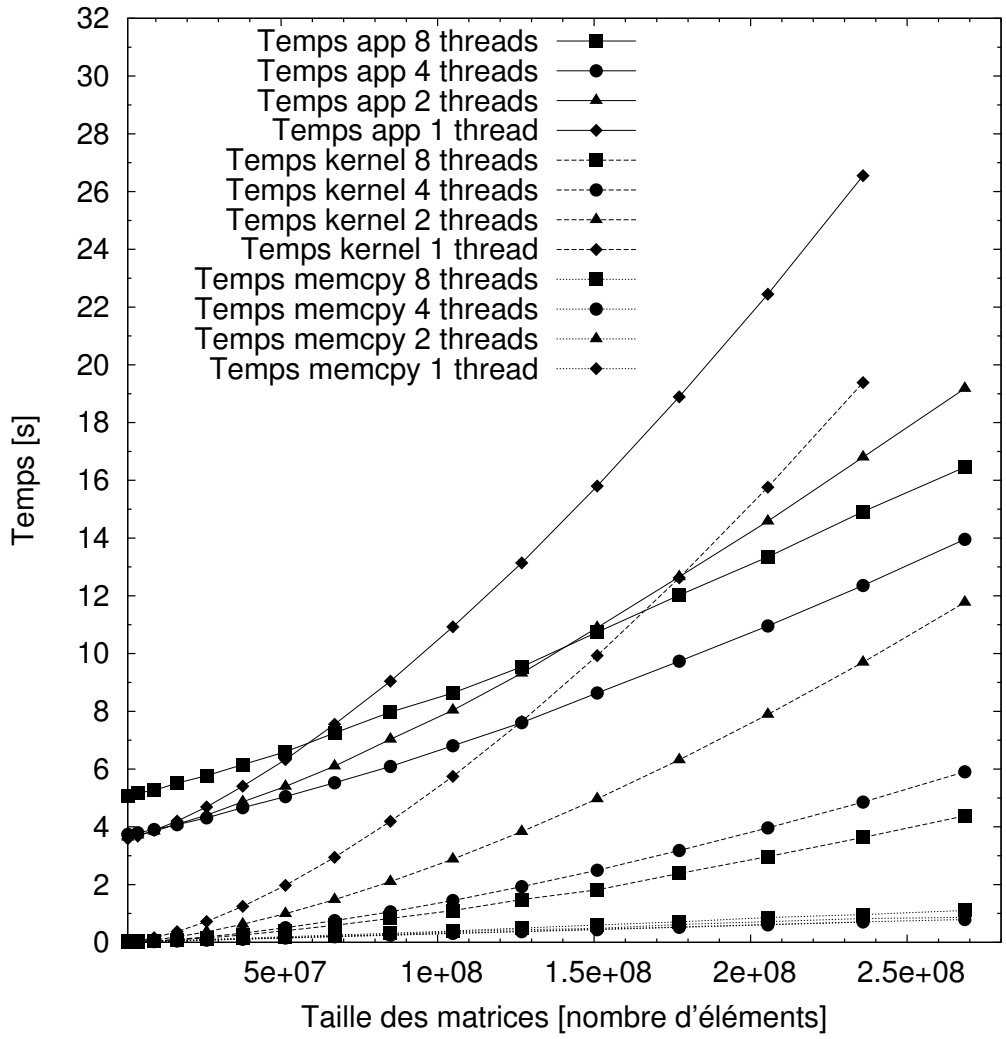


Figure IV.10: Temps d'exécution pour MPI avec cudaHostAlloc (mémoire fixée)

est limitée par la taille de la mémoire vive des CPU, ce qui empêche donc d'aborder des problèmes de très grande taille (voir les limites, en partie 6)) : pour de telles instances dimensionnantes des problèmes, il faut distribuer la mémoire entre plusieurs nœuds multiGPU ou se limiter à une allocation standard.

5) Recouvrement des transferts mémoire CPU-GPU

Nos expériences ont montré que ce problème n'est pas limité par les temps d'accès mémoire, dans la mesure où les temps kernel sont bien plus importants que les temps de transfert des données. La figure IV.12 présente les temps d'exécution des kernels et les temps de transfert mémoire (échelle logarithmique) dans le cas d'OpenMP avec `cudaHostAlloc`.

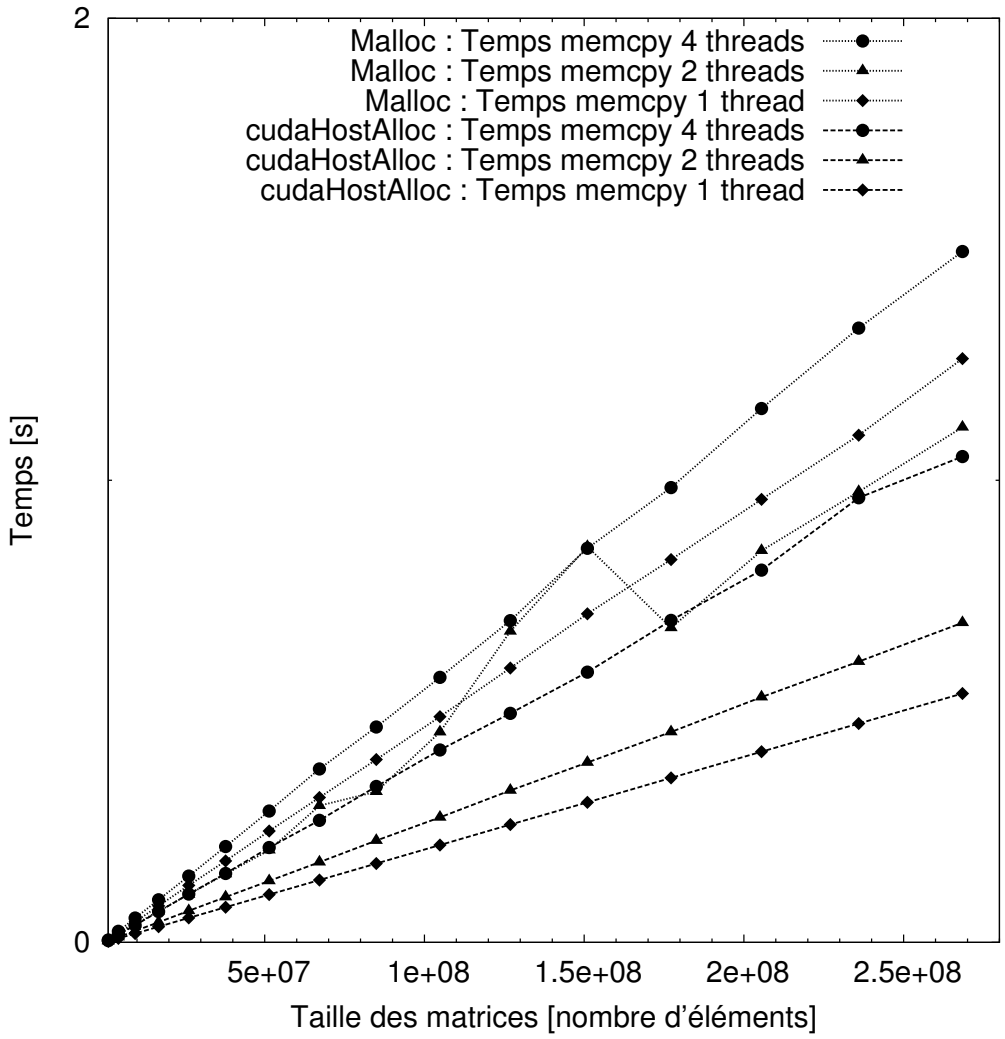


FIGURE IV.11: Temps copie mémoire pour OpenMP avec `cudaHostAlloc` (lignes brisées) et `Malloc` (lignes pointillées)

Ainsi, dans le cas de ce problème cible, il n'est pas essentiel de mettre en œuvre une stratégie de recouvrement des communications CPU-GPU par les calculs GPU. Cependant, certaines applications pouvant être *memory bound*, il faut systématiquement considérer le bénéfice qu'apporterait ainsi l'utilisation de *streams* CUDA.

Un *stream* (flux) CUDA représente une file d'opérations qui seront exécutées par le GPU de façon séquentielle, que ces opérations soient synchrones ou non. Les opérations en questions consistent en des appels kernel, des transferts mémoire, des lancements ou arrêts d'événements. Lorsque deux streams contiennent des opérations asynchrones, celles-ci peuvent être exécutées en parallèle, comme le montre la figure IV.13 qui présente un schéma d'exécution de deux streams CUDA : la première copie mémoire du stream 2, s'exécute en même temps que le kernel du stream 1, et la deuxième copie mémoire du *stream* 1 s'exécute en même temps que le kernel du *stream* 2, ce qui permet

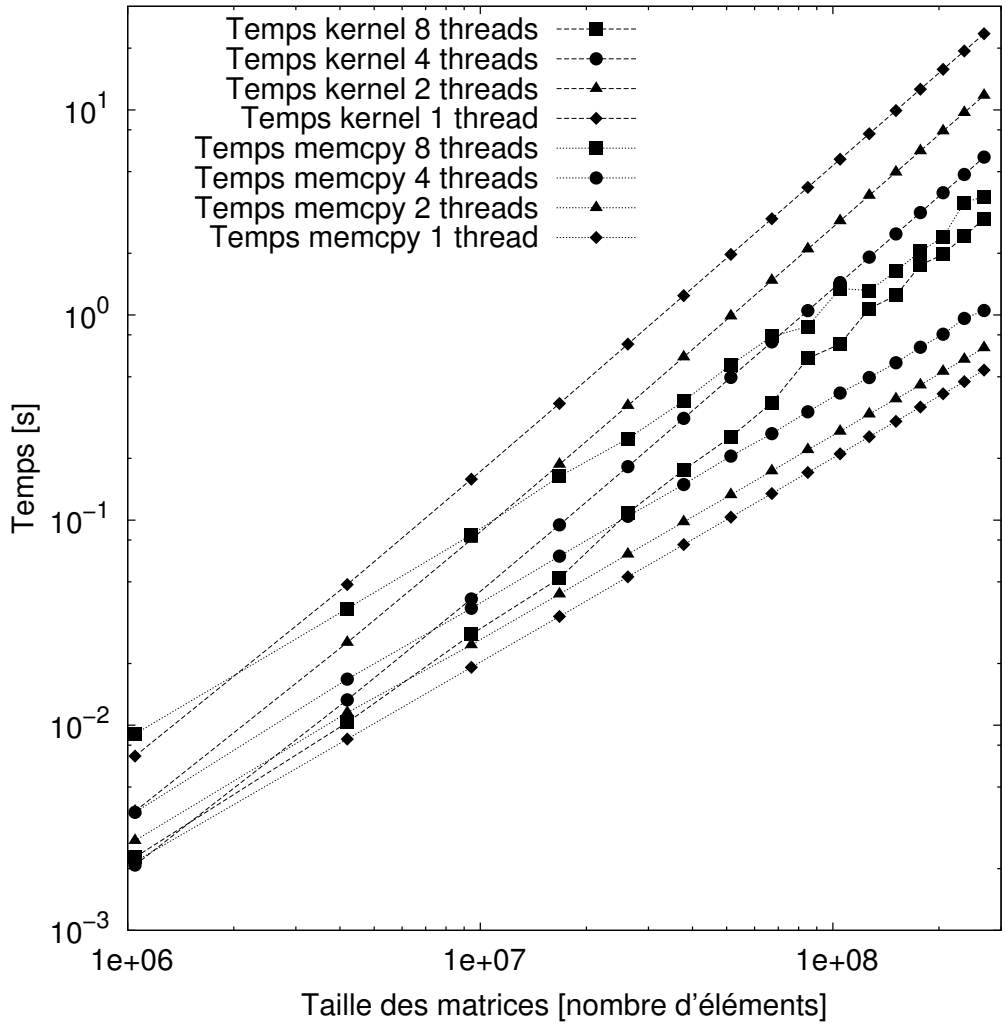


Figure IV.12: Temps copie mémoire et kernel pour OpenMP avec `cudaHostAlloc` (mémoire fixée)

un recouvrement des temps de transfert mémoire [d'un des streams] par les temps de calcul [d'un autre stream].
Les principaux appels CUDA sont synchrones mais sont doublés de versions asynchrones qu'il est possible d'utiliser dans des streams. C'est le cas des appels de kernel, qui sont nécessairement asynchrones, et des fonctions de copie mémoire, dont la version `cudaMemcpyAsync` est asynchrone [3]. On notera que, pour pouvoir utiliser des copies asynchrones, les adresses mémoire utilisées doivent être dans des zones de mémoire non

3. La fonction `cudaMemcpy`, qui se comporte comme la fonction `memcpy` de la bibliothèque standard C, est synchrone. `cudaMemcpyAsync` en est une version asynchrone : elle « revient » de l'appel aussitôt, sans que la copie ait démarré et encore moins qu'elle soit terminée ; cependant, comme les différents appels sont placés dans des flux, nous pouvons être certains que la copie aura entièrement eu lieu avant l'opération suivante du même flux.

paginable.
Ainsi, pour les cas d'applications dont les temps de transfert mémoire peuvent être significatifs, l'utilisation de streams CUDA permet d'éviter que ces temps d'accès soient pénalisants pour les temps globaux d'exécution de l'application.

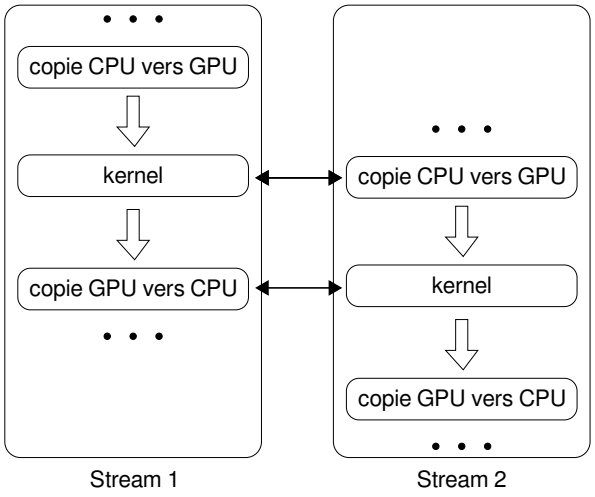


Figure IV.13: Exécution de deux streams CUDA

6) Limites des applications et accélérations

Afin de donner une idée claire sur les performances obtenues, le tableau IV.1 présente les données numériques des temps d'exécution des applications, et les accélérations correspondantes, pour les implémentations basées sur OpenMP et MPI ; nous y donnons les résultats obtenus avec 1 à 8 threads, en fonction des dimensions des matrices. On notera que nous ne considérons ici que le cas [favorable] de l'allocation en mémoire fixée (`cudaHostAlloc`).

On peut observer que la taille des matrices envisageables est limitée. La première limite est imposée par la quantité de mémoire disponible sur GPU, qui est assez faible (4 Go par carte graphique pour les modèles considérés). Un deuxième niveau de limites est imposé par la mémoire fixée disponible, ressource limitée par la quantité totale de mémoire vive disponible sur le système (ici 8 Go).
Nous constatons également qu'il est possible de considérer des problèmes de plus grande taille avec l'implémentation OpenMP. En effet, l'implémentation MPI utilise une quantité légèrement supérieure de mémoire puisque, pour faire circuler les sous-matrices de B entre les processus, nous devons dupliquer l'espace mémoire nécessaire au stockage de ces sous-matrices « cyclées ».

Les courbes précédentes montrent que les modèles proposés permettent un bon passage à l'échelle, particulièrement pour l'approche OpenMP-HostAlloc, jusqu'aux limites de la mémoire des GPU (voir la figure IV.9). Pour compléter l'analyse de ce comportement, nous pouvons faire une comparaison quantitative globale des résultats.

MPI – Nombre des processus lourds									OpenMP – Nombre des processus légers						
1	2		4		8		*dim.*	*taille*	1	2		4		8	
Temps	Temps	Accél.	Temps	Accél.	Temps	Accél.	*mat.*	*données*	Temps	Temps	Accél.	Temps	Accél.	Temps	Accél.
3,59	3,64	0,98	3,74	0,96	5,08	0,71	*1024*	*[1]*	3,58	3,73	0,96	4,03	0,89	4,64	0,77
3,68	3,71	0,99	3,80	0,97	5,18	0,71	*2048*	*[4]*	3,64	3,77	0,97	4,06	0,90	4,68	0,78
4,20	4,09	1,03	4,08	1,03	5,52	0,76	*4096*	*[16]*	4,06	4,00	1,02	4,24	0,96	4,91	0,83
5,41	4,86	1,11	4,66	1,16	6,15	0,88	*6144*	*[36]*	5,07	4,53	1,12	4,58	1,11	5,30	0,96
7,55	6,10	1,24	5,53	1,37	7,25	1,04	*8192*	*[64]*	6,98	5,61	1,25	5,20	1,34	6,00	1,16
10,93	8,04	1,36	6,81	1,60	8,63	1,27	*10240*	*[100]*	10,05	7,32	1,37	6,16	1,63	7,04	1,43
15,80	10,90	1,45	8,63	1,83	10,74	1,47	*12288*	*[144]*	14,56	9,78	1,49	7,57	1,92	8,49	1,72
22,45	14,58	1,54	10,96	2,05	13,34	1,68	*14336*	*[196]*	20,78	13,13	1,58	9,55	2,18	10,48	1,98
28,97	19,18	1,51	13,96	2,08	16,45	1,76	*16384*	*[256]*	28,97	17,54	1,65	12,08	2,40	12,94	2,24
35,58	25,23	1,41	17,57	2,02	20,72	1,22	*18432*	*[324]*	39,42	23,08	1,71	15,31	2,57	16,14	2,44
	33,08		21,96		25,60		*20480*	*[400]*		29,95		19,24		20,23	
							22528	*[484]*		38,28		23,85			
							24576	*[576]*				29,40			

TABLE IV.1: Temps d'exécution et accélérations, 1 à 8 threads/processus, en fonction de la taille du problème (`cudaHostAlloc`)

– L'observation des accélérations basées sur les temps globaux (tableau IV.2) montre par ailleurs un avantage d'environ 27% pour l'approche OpenMP contre celle basée sur MPI, pour les plus grands problèmes, lorsque le serveur Tesla est utilisé d'une façon optimale (4 processus légers/lourds contrôlant les 4 cartes GPU). Cela s'explique par le fait que la gestion des contextes est moins lourde en mémoie partagée qu'avec des échanges de messages (création et gestion allégée).

Taille matrice	1024	2048	4096	6144	8192	10240	12288	14336	16384	18432
OpenMP	0,89	0,90	0,96	1,11	1,34	1,63	1,92	2,18	2,40	2,57
MPI	0,96	0,97	1,03	1,16	1,37	1,60	1,83	2,05	2,08	2,02

TABLE IV.2: Accélérations (4 threads/processus *vs.* 1 thread/processus)

– Par ailleurs, comme les temps globaux sont presque identiques dans la configuration un thread ou processus par carte, il est possible de comparer les efficacités obtenues pour les applications codées avec OpenMP et MPI (toujours avec 4 threads/processus pour gérer autant de cartes GPU : Figure IV.3. Dans ce domaine nous constatons à nouveau un avantage pour l'implémentation en mémoire partagée, avec une efficacité de 64% (50,5% pour MPI) pour des matrices de taille 18432x18432, lorsqu'on contrôle les 4 GPU avec 4 processus légers [4].

Taille matrice	1024	2048	4096	6144	8192	10240	12288	14336	16384	18432
OpenMP	22%	23%	24%	28%	33,5%	41%	48%	54,5%	60%	64%
MPI	24%	24%	26%	29%	34%	40%	46%	51%	52%	50,5%

TABLE IV.3: Efficacités (4 threads/processus *vs.* 1 thread/processus)

7) Performance du modèle de gestion multiGPU

Comme nous venons de le noter, l'implémentation en mémoire partagée de notre plateforme est plus naturelle et donne de meilleures performances que son équivalent implémenté par passage de message. Dans cette partie nous detaillons donc les résultats obtenus avec OpenMP+`cudaHostAlloc` (allocation en mémoire *figée*, qui elle aussi est la plus favorable en terme de performance des accès mémoire).

Les caractéristiques du problème de benchmark (cf IV.2.1) nous permettent de déterminer la puissance effective tirée des GPU. Le tableau IV.4 présente les performances [5]

4. Aucune efficacité ne peut être donnée pour les plus grandes envisageables, car il a été impossible de les traiter avec un seul GPU.

5. puissance : nombre d'opérations de calcul par unité de temps

obtenues en utilisant la pleine puissance du nœud de calcul (4 threads pour gérer les 4 GPU du serveur Tesla). Nous les y donnons en fonction de la taille problème, avec l'objectif de comparer la puissance globale (obtenue sur la base des temps de calcul globaux) et la puissance de calcul (en partant des *temps kernel*, temps de calcul effectif).

Taille matrice n (x1024)	Opérations n^3 (x1024^3)	Temps application (s)	Performance (GFlops)	Temps kernel (s)	Puissance effective (GFlops)	Pourcentage %
1	1	4,03	0,27	0,002	516,22	0,05
2	8	4,06	2,12	0,013	645,67	0,33
4	64	4,24	16,21	0,095	724,33	2,24
6	216	4,58	50,64	0,313	740,32	6,84
8	512	5,20	105,72	0,739	743,72	14,22
10	1000	6,16	174,31	1,439	746,02	23,37
12	1728	7,57	245,10	2,484	746,92	32,81
14	2744	9,55	308,52	3,945	746,84	41,31
16	4096	12,08	364,08	5,886	747,16	48,73
18	5832	15,31	409,02	8,375	747,67	54,71
20	8000	19,24	446,46	11,488	747,70	59,71
22	10648	23,85	479,38	15,292	747,67	64,12
24	13824	29,40	504,88	19,850	747,78	67,52

TABLE IV.4: Performances atteintes (puissances globale ; puissance de calcul) en fonction de la taille du problème – implémentation OpenMP-`cudaHostAlloc`

– Tout d'abord nous pouvons observer que la performance globale obtenue est en constante augmentation avec la taille du problème, pour aboutir à plus de 500 GFlops soutenus.
– En limitant l'évaluation des temps aux temps kernel (temps de calcul effectif), on note une puissance effective constamment comprise entre 746 et 748 GFlops à partir de problèmes de taille suffisante, ce qui montre que les calculs ne sont pas entravés par des synchronisations, et donc que notre modèle permet de tirer toute la puissance possible des cartes graphiques.
  Il s'avère que nous n'atteignons *que* 20% de la puissance crête théorique annoncée par le constructeur [6]. Vue la constance observée pour les performances de calcul sur notre application cible, nous pouvons affirmer que cette performance relative n'est pas liée au modèle de gestion multi-carte, mais à l'application de benchmark elle-même et aux optimisations réalisées sur son code.
– Lorsque la taille du problème augmente, on constate que la proportion de calcul effectif dans le temps de calcul global augmente (plus de 67% pour les plus grands

6. La puissance crête théorique du nœud Tesla T10 est de 3,74 TFlops (4 cartes à 936 GFlops).

problèmes), et donc que l'incidence des transferts mémoire dans le temps de calcul global diminue. Comme le présente la courbe de la figure IV.14 cette évolution, qui quantifie la pertinence du modèle de gestion des nœuds multiGPU, offre de perspectives plus qu'encourageantes. Pour les réaliser il faudra dépasser les capacités de calcul d'un unique nœud multiGPU et passer à des grappes de multiGPU, avec un modèle disposant d'une dimension additionnelle de communication inter-nœuds (par échange de messages).

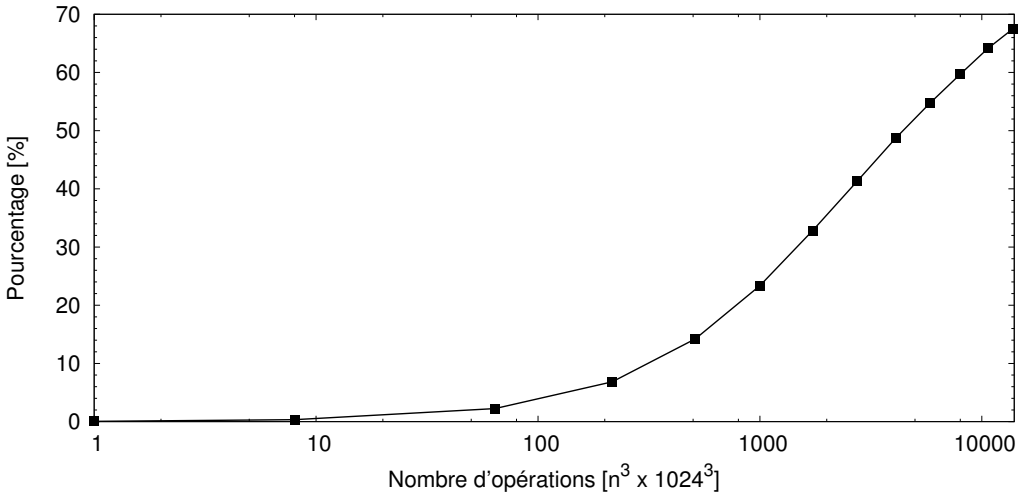


Figure IV.14: Efficacité du modèle de gestion de nœuds multiGPU, en fonction de la taille du problème (nombre d'opérations, échelle logarithmique)

Ainsi ces considérations valident l'intérêt de ce modèle de gestion des nœuds multiGPU. Nous l'étendrons prochainement en un modèle à trois niveaux, applicable pour répartir du calcul sur les nœuds de calcul d'une architecture constituée de plusieurs nœuds multiGPU. Couplé avec notre outil de transformation automatique de code OpenMP en code CUDA (voire à un outil permettant de générer automatiquement du code parallèle à partir de code séquentiel), nous devrions pouvoir disposer finalement d'une plateforme complète permettant de tirer parti des futures architectures exascale.

IV.4 Conclusion

Nous avons vu dans ce chapitre que le modèle de programmation en mémoire partagée constitue un bon choix pour gérer les environnements multi-cartes au sein d'un nœud de calcul. Les avantages en sont la facilité de programmation ainsi qu'une meilleure performance (gestion allégée des contextes GPU par des threads).
Sur la base de notre outil de transformation de code OpenMP en code CUDA (voir le chapitre précédent), cette étude montre qu'il est possible de l'étendre à un nœud multi-cartes, puis de le généraliser à une approche multi-niveaux : utiliser le code

CUDA généré pour décrire l'exécution des calculs sur chaque carte graphique, utiliser la programmation en mémoire partagée (OpenMP) pour gérer plusieurs cartes à l'intérieur d'un nœud et utiliser la programmation par échange de messages (MPI) pour les communication entre les nœuds. Les trois modèles de programmation étant orthogonaux, ils peuvent en effet être combinés sans difficulté.

Cette approche multi niveaux devrait permettre, à moyen et long terme, d'assurer une bonne maîtrise de la programmation des supercalculateurs exascale qui disposeront de milliers d'accélérateurs. De plus l'évolution rapide de la technologie devrait encore apporter de nouvelles avancées pour faciliter l'effort de programmation multi niveaux. Pour aller dans ce sens, les versions 4 et 5 du SDK CUDA intègrent des fonctionnalités dont nous ne disposions pas au moment de notre étude, et qui devraient servir nos projets :

- Nous pouvons en particulier évoquer le *GPU Direct Peer-to-Peer*, qui permet une communication directe des cartes graphiques connectées sur le même bus PCIe. Ce mécanisme a été rendu possible par une autre amélioration, nommée *Unified Virtual Addressing*, qui permet d'avoir un espace d'adressage commun entre CPU et GPU.
- L'amélioration du driver et du framework CUDA a permis de lever d'autres limitations. En particulier il est désormais possible à la fois d'attacher plusieurs threads à un seul GPU et aussi de gérer plusieurs GPU avec un seul thread.
- Au niveau des communications inter-nœuds une autre innovation, nommé *GPU Direct RDMA*, améliorera d'un manière significative les échanges de messages MPI. En effet les transferts ne passeront plus par le CPU mais seront acheminés directement par le module DMA.

  Une autre fonctionnalité récente devrait améliorer l'utilisation de MPI entre nœuds GPU ou multi-GPU, puisque le matériel intègre désormais une logique de contrôle capable de gérer jusqu'à 32 queues de tâches MPI par carte (technologie *Hyper-Q*). Cette nouveauté n'est actuellement disponible que pour les cartes NVIDIA Kepler haut de gamme (GK110) mais devrait elle aussi être étendue.

Les nouveautés mises en place par NVIDIA pour gérer un ensemble de cartes graphiques confirment l'intérêt de notre approche multiGPU. D'une part elles mettent en évidence la nécessité de combler le manque auquel nous avons répondu ; d'autre part elles devraient contribuer à simplifier la mise en œuvre de nos stratégies de répartition des calculs, et également permettre d'envisager des performances plus prometteuses en vue d'exploiter les supercalculateurs exascale qui nous sont promis dans un avenir proche (voir I.2 : *Évolution des machines parallèles*).

Conclusion et perspectives

Les architectures hybrides many-cœurs ont trouvé leur place dans le paysage HPC actuel. Elles sont considérées comme un des moyens incontournables pour franchir la barre de l'exascale dans les années à venir. Cependant leur modèle de programmation est neuf et difficile à utiliser, d'où le besoin de plus en plus important d'outils pour tirer profit des performances théoriques annoncées.

En même temps des milliers d'anciennes lignes de code utilisent le modèle de programmation OpenMP, basé sur des directives, qui a prouvé son efficacité au fil du temps et qui est aujourd'hui l'un des modèles de programmation les plus utilisés pour exprimer le parallélisme d'une application.

Cette thèse s'inscrit dans le cadre de l'étude des méthodes de programmation pour les architectures hybrides. Notre travail a pour but de proposer un environnement parallèle de développement haut niveau pour les accélérateurs graphiques. Pour arriver à ce but précis nous avons suivi deux axes de recherche complémentaires à des niveaux différents :

– pour obtenir du code spécifique pour un accélérateur, nous avons proposé une méthode de transformation automatique de code, permettant de partir de code de haut niveau écrit en OpenMP afin de le transformer en un code de bas niveau, équivalent, capable d'être exécuté sur des cartes graphiques NVIDIA, mais suffisamment lisible pour qu'il soit possible de le retravailler pour des optimisations ultérieures ;
– toujours au sein d'un nœud GPU nous avons étudié et mis en place des schémas d'exécution appropriés pour gérer plusieurs cartes graphiques, dans le but de résoudre des problèmes de plus grande taille.

Afin de mener notre démarche scientifique, le premier chapitre de la thèse présente un état de l'art des modèles d'exécution parallèle, ainsi que différentes taxonomies pour bien intégrer les accélérateurs. Nous analysons ensuite l'évolution récent du TOP500, qui met en évidence la contribution des machines hybrides pour atteindre l'exascale car les accélérateurs et notamment les cartes graphiques, tout d'abord banalisées dans les ordinateurs grand public, ont maintenant toute leur place dans la constitution des machines les plus puissantes au monde. Pour toute démarche concernant leur programmabilité, une connaissance approfondie des architectures bas niveau est nécessaire pour pouvoir tirer le maximum de performance. Pour cette raison nous avons consacré la fin de chapitre I à une présentation détaillée des particularités architecturales des accélérateurs les plus représentatifs à ce jour (les cartes graphiques : NVIDIA et AMD ;

les accélérateurs spécialisés : ClearSpeed, CellBE, FPGA ; ainsi que le tout récent Intel Xeon Phi).

Le chapitre II présente en détails les différents langages disponibles pour programmer les accélérateurs. Nous commençons bien évidemment par les modèles génériques de programmation, pour continuer avec les langages bas niveau spécifiques aux cartes graphiques. Ces langages, dits de bas niveau car ils sont le plus souvent liés à une architecture spécifique, ont beaucoup évolué et n'ont désormais plus rien à voir avec les anciens langages de programmation graphique. Les nouveaux langages sont proches du langage C et en étendent les possibilités. Afin d'améliorer l'adoption des accélérateurs, de nouveaux langages de haut niveau ont fait leur apparition. Nous constatons que la tendance est d'utiliser des langages basés sur des directives car ils sont faciles à utiliser pour les développeurs ; notons cependant qu'ils déplacent le challenge vers les compilateurs, qui doivent être capables de générer du parallélisme automatiquement. Nous avons consacré la dernière partie de ce chapitre à un état de l'art des outils existant, capables de transformer du code haut niveau en du code bas niveau. Cela nous a permis de mieux encadrer notre travail dans le contexte actuel et, en même temps, de valider notre approche.

La deuxième partie de cette thèse est consacrée exclusivement aux contributions personnelles. Le chapitre III présente notre démarche pour créer un environnement parallèle de développement haut niveau pour les accélérateurs graphiques. Sans perdre de généralité nous avons choisi de générer du code bas niveau CUDA pour les cartes graphiques NVIDIA à partir du code du haut niveau écrit avecOpenMP. Néanmoins, nous nous sommes limités aux portions du code OpenMP vraiment intéressantes que sont les boucles parallèles, car elles sont susceptibles de profiter d'un gain important en étant déportées sur une carte graphique. En effet les cartes graphiques ont des architectures SIMT qui sont particulièrement adaptées au parallélisme de données. Nous avons utilisé comme base le transformateur OMPi, capable de générer des threads POSIX à partir de directives OpenMP, que nous avons modifié pour prendre en charge le langage CUDA et être capables de générer du code dans ce langage. La transformation systématique du code, qui se déroule intégralement au sein de l'arbre syntaxique, permet ainsi de prendre en charge ce nouveau langage comme code final. Notre outil supporte un sous-ensemble du modèle OpenMP, plus spécifiquement les directives `omp parallel for` avec des boucles simples ou imbriquées. Dans ce chapitre III nous avons donc détaillé notre outil et les étapes de chaque transformation. Nous dégageons alors des pistes immédiates pour l'amélioration de notre outil : certaines sont décrites en détails et seront implémentées prochainement, comme le support des directives `critical` et `single` ou des clauses `firstprivate` et `lastprivate`. Une partie entière est dédiée aux optimisations envisageables du code généré : en effet dans un premier temps nous avons cherché la correctitude du code produit, mais nous envisageons notamment le *tunning* automatique du code généré, par rapport au matériel sur le quel le code s'exécutera, basé sur des réglages des paramètres d'exécution (nombre de threads par bloc, le nombre de bloc par grille).

Le dernier chapitre est consacré à une étude que nous avons menée pour gérer plusieurs cartes graphiques à l'intérieur d'un seul nœud, car cette configuration devient de plus en plus courante dans les stations de travail et dans les nœuds des supercalculateurs hybrides. Cette étude nous a révélé que la programmation en mémoire partagée est là encore le moyen le plus intéressant pour gérer plusieurs GPU, tant du point de vue de la simplicité de l'implémentation que pour la gestion des ressources utilisées.

Les perspectives qui se dégagent de nos travaux sont nombreuses et s'expriment dans les deux domaines de recherche de cette thèse :

- À court terme, implémenter les optimisations présentés dans le chapitre III ainsi que supporter plusieurs directives OpenMP comme par exemple la directive `parallel section` (qui peut être transformée en un kernel à part).
- À moyen terme, nous envisageons a créer un outil intégré capable de générer du parallélisme pour les cartes graphiques à partir de code C standard. Une collaboration est en cours sur ce thème avec l'équipe du groupe Alchemy/Grand Large de l'Université Paris Sud et nous envisageons de mettre nos efforts en commun pour aboutir à un outil intégrant la capacités d'auto-parallélisation de PoCC [PBB+10], la transformation du code généré (code C enrichi de directives de type OpenMP) en code dédié pour GPU et l'optimisation pour une exécution sur une architecture spécifique (en fonction de ses caractéristiques).
- Dans l'autre axe de recherche, pour permettre l'exécution de codes parallèles sur un ensemble de nœuds hybrides nous proposons de développer un outil multi-niveaux capable d'utiliser tous les moyens disponibles dans une grappe de plusieurs nœuds, chacun disposant de plusieurs GPU. L'enjeu est de proposer un concept de programmation de haut niveau, optimisé pour les architectures cibles. Le schéma du modèle, à trois niveaux, consiste à réaliser les communications inter-nœuds via des processus MPI (mémoire distribuée) ; à prendre en charge un nœud multiGPU à l'aide de threads OpenMP en mémoire partagée, pour que les calculs soient repartis sur les différentes cartes GPU et, enfin, exécutés au sein des GPU eux-mêmes. La mise en œuvre de ce modèle devrait permettre d'adresser les prochaines machines exascale.

Annexe 1 : Les fichiers du transformateur de code OMPi

Les fichiers les plus importantes du transformateur de code OMPi sont les suivants :

ompi.c : est le fichier principale qui parmi d'autre fonctionnalités, prend en charge les appels du parser et les fonctions de transformations.

scanner.l, parser.y : Les expressions réguliers de l'analyseur lexicale sont implémenté dans le fichier `scanner.l`, tandis que la grammaire de l'analyseur syntaxique est implémenté dans le fichier `parser.y`. La grammaire génère aussi l'arbre syntaxique.

ast.c, symtab.c : Contient les fonctions principales pour la pris en charge de l'arbre syntaxique et du tableau des symboles.

ast_show.c, ast_print.c, ast_free.c, ast_copy.c : Ces fichiers prennent en charge l'affichage à la sortie standard ou dans un chaîne du caractères du AST ou d'une sous-partie, l'eliberations des ressources ou une copie intégrale ou partielle.

ast_xform.c : Contient les fonctions qui dirige tout les transformations, ainsi que l'implémentation des transformations simples.

x_parallel.c, x_single.c, x_sections.c, x_for.c, x_thrpriv.c : Contient les implémentations des transformations complexes pour les directives `parallel`, `for`, `single`, `sections`, `threadprivate`.

ast_vars.c, x_clauses.c : Contient l'analyse et les transformations pour les variables générique, ainsi que celle dans les clause OpenMP.

La figure A.1 présente un graphe simplifié des dépendances des fichiers sources du projet OMPi. Les nœuds gris représentent les fichiers que nous avons modifiés pour supporter les transformations CUDA.

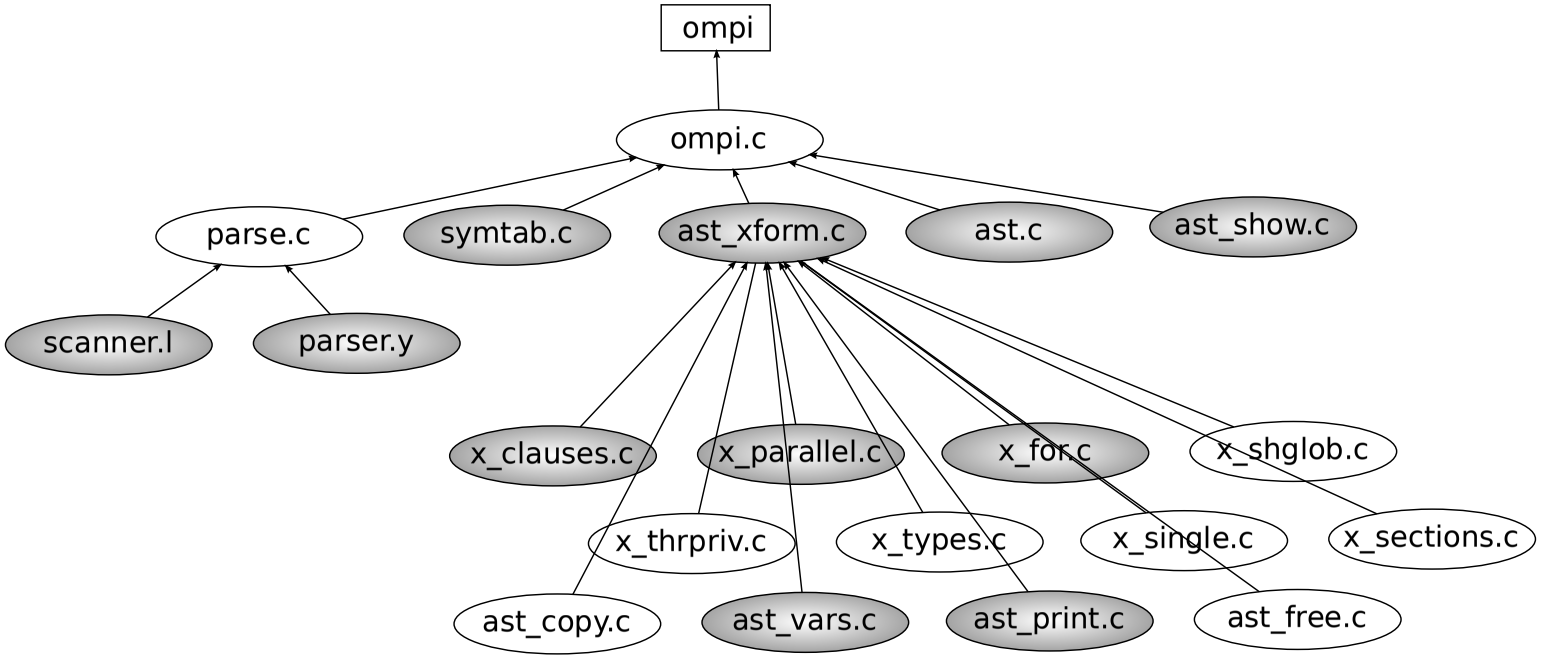


Figure A.1: Projet OMPi : graphe simplifié des dépendances des fichiers source

Références

[ABB+10] E. Ayguadé, R.M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, et al. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5) :440–459, 2010.

[ACC+96] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS : a Workbench for Program Parallelization and Optimization. European Parallel Tool Meeting (EPTM), Octobre 1996. Onera, France.

[Adv12] Advanced Micro Devices. *AMD Graphics Cores Next (GCN) Architecture Whitepaper*. AMD, Juin 2012.

[AMD10] AMD Streamcomputing website `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`, Juillet 2010.

[ATI09] ATI Introduction to OpenCL website `http://ati.amd.com/technology/streamcomputing/intro_opencl.html`, Septembre 2009.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer Berlin / Heidelberg, 2010.

[BSHdS11] J. Beyer, E. Stotzer, A. Hart, and B. de Supinski. OpenMP for accelerators. *OpenMP in the Petascale Era*, pages 108–121, 2011.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys (CSUR)*, 21(3) :261–322, 1989.

[CDK+00] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[cet09] Cetus website `http://cetus.ecn.purdue.edu/`, Septembre 2009.

[CJvdPK07] Barbara Chapman, Gabriele Jost, Ruud van der Pas, and David J. Kuck. *Using OpenMP : Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[Cle08] ClearSpeed. CSX700 Floating Point Processor Datasheet. Technical report, 8 2008.

[CPJ10] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias MÃ$\frac{1}{4}$ller, Barbara Chapman, and Bronis de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin / Heidelberg, 2010.

[CUD] CUDA Occupancy Calculator, Septembre, 2012.

[DBB07] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of GPGPU, First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, Ma, USA, 2007.

[DLP03] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark : past, present and future. *Concurrency and Computation : Practice and Experience*, 15(9) :803–820, 2003.

[DLT03] Vassilios Dimakopoulos, Elias Leontiadis, and George Tzoumas. A portable C compiler for OpenMP V.2.0. In *EWOMP 2003, European Workshop on OpenMP*, 2003.

[Don79] J.J. Dongarra. *LINPACK : Users' Guide*. Number 8. Society for Industrial Mathematics, 1979.

[Dun90] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2) :5 –16, feb. 1990.

[FC07] W. Feng and K.W. Cameron. The green500 list : Encouraging sustainable supercomputing. *Computer*, 40(12) :50–55, 2007.

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

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, 1972.

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

[FS09] W. Feng and T. Scogland. The green500 list : Year one. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–7. IEEE, 2009.

[Gre11] K. Gregory. Overview and C++ AMP approach. Technical report, Technical report, Microsoft, Providence, RI, USA, 2011.

[Int12] Intel Xeon Phi website `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`, Septembre 2012.

[ISO99] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899 :1999 draft.

[Joh88] E.E. Johnson. Completing an MIMD multiprocessor taxonomy. *ACM SIGARCH Computer Architecture News*, 16(3) :44–47, 1988.

[KB04] S. Karlsson and M. Brorsson. A free OpenMP compiler and run-time library infrastructure for research on shared memory parallel computing. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 354–361, 2004.

[KDH+05] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM journal of Research and Development*, 49(4.5) :589–604, 2005.

[ker12a] KernelGen a toolchain for automatic GPU-centric applications porting, Août 2012.

[ker12b] Kernelgen website `http://kernelgen.org`, Octobre 2012.

[KH10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann Publishers, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, 2004.

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU : a compiler framework for automatic translation and optimization. In *PPoPP '09 : Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla : A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2) :39–55, 2008.

[LQPdS10] Chunhua Liao, Daniel Quinlan, Thomas Panas, and Bronis de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias Mï¿½ller, Barbara Chapman, and Bronis de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin / Heidelberg, 2010.

[LRDG90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time robot motion planning using rasterizing computer graphics hardware*, volume 24. ACM, 1990.

[MDK12] Rama Malladi, Richard Dodson, and Vyacheslav Kitaeff. Intel Many Integrated Core (MIC) Architecture : Portability and Performance Efficiency

Study of Radio Astronomy algorithms. In *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, Astro-HPC '12, pages 5–6, New York, NY, USA, 2012. ACM.

[Mes05] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard*. University of Tennessee, 15/11/2005.

[Moo65] Gordon Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

[Moo75] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13. IEEE, 1975.

[MWBA10] R.C. Murphy, K.B. Wheeler, B.W. Barrett, and J. Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, 2010.

[ND10] John Nickolls and William J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2) :56–69, 2010.

[NJK11] G. Noaje, C. Jaillet, and M. Krajecki. Source-to-Source Code Translator : OpenMP C to CUDA. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 512 –519, sept. 2011.

[NKJ10] G. Noaje, M. Krajecki, and C. Jaillet. MultiGPU computing using MPI or OpenMP. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 347 –354, aug. 2010.

[NVIa] NVIDIA Corporation. *CUBLAS Library Version 5.0*. NVIDIA Corporation, Santa Clara, CA, USA.

[NVIb] NVIDIA Corporation. *CUDA Best Practices Guide Version 5.0*. NVIDIA Corporation, Santa Clara, CA, USA.

[NVIc] NVIDIA Corporation. *CUDA Programming Guide Version 5.0*. NVIDIA Corporation, Santa Clara, CA, USA.

[NVId] NVIDIA Corporation. *Fermi Architecture Whitepaper*. NVIDIA Corporation.

[NVIe] NVIDIA Corporation. *Kepler GK110 Architecture Whitepaper*. NVIDIA Corporation.

[Ope08] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2008.

[ope09a] OpenCL for MacOS X Snow Leopard website `http://www.apple.com/macosx/`, Septembre 2009.

[ope09b] OpenCL for NVIDIA website `http://www.nvidia.com/object/cuda_opencl.html`, Septembre 2009.

[Ope10] OpenCL website `http://www.khronos.org/registry/cl/`, Juillet 2010.

[Ope12] OpenACC website `http://www.openacc-standard.org/`, Septembre 2012.

[PBB+10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework . In *Conference on Supercomputing (SC10)*, New Orleans, LA, nov 2010. IEEE Computer Society Press.

[PG08] The Portland Group. *PGI Fortran & C Accelerator Programming Model*. Portland Group, The, Decembre 2008.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[Rao09] Benoit Raoult. HMPP Workbench - Build manycore applications. In *GPU Accelerators and Hybrid Computing Workshop*, Juillet 2009.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3) :1–15, 2008.

[SHI00] M. Sato, H. Harada, and Y. Ishikawa. Openmp compiler for a software distributed shared memory system scash. *WOMPAT2000*, 2000.

[Sør12] H.H.B. Sørensen. Auto-tuning of level 1 and level 2 BLAS for GPUs. *Concurrency and Computation : Practice and Experience*, 2012.

[ST98] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2) :123–169, 1998.

[Vol10] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[Wol08] Michael Wolfe. Compilers and More : A GPU and Accelerator Programming Model. HPCWire, Decembre 2008.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. *SIGOPS Oper. Syst. Rev.*, 21(5) :63–76, Novembre 1987.

UN ENVIRONNEMENT PARALLÈLE DE DÉVELOPPEMENT HAUT NIVEAU POUR LES ACCÉLÉRATEURS GRAPHIQUES : MISE EN ŒUVRE À L'AIDE D'OPENMP

Les processeurs graphiques (GPU), originellement dédiés à l'accélération de traitements graphiques, ont une structure hautement parallèle. Les innovations matérielles et de langage de programmation ont permis d'ouvrir le domaine du GPGPU, où les cartes graphiques sont utilisées comme des accélérateurs de calcul pour des applications HPC généralistes. L'objectif de nos travaux est de faciliter l'utilisation de ces nouvelles architectures pour les besoins du calcul haute performance ; ils suivent deux objectifs complémentaires. Le premier axe de nos recherches concerne la transformation automatique de code, permettant de partir d'un code de haut niveau pour le transformer en un code de bas niveau, équivalent, pouvant être exécuté sur des accélérateurs. Dans ce but nous avons implémenté un transformateur de code capable de prendre en charge les boucles « pour » parallèles d'un code OpenMP (simples ou imbriquées) et de le transformer en un code CUDA équivalent, qui soit suffisamment lisible pour permettre de le retravailler par des optimisations ultérieures. Par ailleurs, le futur des architectures HPC réside dans les architectures distribuées basées sur des nœuds dotés d'accélérateurs. Pour permettre aux utilisateurs d'exploiter les nœuds multiGPU, il est nécessaire de mettre en place des schémas d'exécution appropriés. Nous avons mené une étude comparative et mis en évidence que les threads OpenMP permettent de gérer de manière efficace plusieurs cartes graphiques et les communications au sein d'un nœud de calcul multiGPU.
Mots clés : OpenMP, CUDA, compilateur, transformation de code, manycoeurs, multiGPU

A HIGH-LEVEL PARALLEL DEVELOPMENT FRAMEWORK FOR GRAPHIC ACCELERATORS : AN IMPLEMENTATION BASED ON OPENMP

Graphic cards (GPUs), initially used for graphic processing, have a highly parallel architecture. Innovations in both architecture and programming languages opened the new domain of GPGPU where GPUs are used as accelerators for general purpose HPC applications. Our main objective is to facilitate the use of these new architectures for high-performance computing needs ; our research follows two main directions. The first direction concerns an automatic code transformation from a high level code into an equivalent low level one, capable of running on accelerators. To this end we implemented a code transformer that can handle parallel "for" loops (single or nested) of an OpenMP code and convert it into an equivalent CUDA code, which is in a human readable form that allows for further optimizations. Moreover, the future of HPC lies in distributed architectures based on hybrid nodes. Specific programming schemes have to be used in order to allow users to benefit from such multiGPU nodes. We conducted a comparative study which revealed that using OpenMP threads is the most adequate way to control multiple graphic cards as well as manage communications efficiently within a multiGPU node.
Keywords : OpenMP, CUDA, compiler, code transformation, manycores, multiGPU