Modèles de programmation et d'exécution pour les architectures parallèles et hybrides. Applications à des codes de simulation pour la physique.

Matthieu Ospici

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Matthieu Ospici

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Thierry Deutsch**

préparée au sein **du Laboratoire d'Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Modèles de programmation et d'exécution pour les architectures parallèles et hybrides

Applications à des codes de simulation pour la physique

Thèse soutenue publiquement le **3 juillet 2013**,
devant le jury composé de :

M. Jean Virieux
Professeur, Université Joseph Fourier, Président
M. Édouard Audit
Professeur, CEA, Rapporteur
M. Pierre Manneback
Professeur, Université de Mons, Rapporteur
M. Dimitri Komatitsch,
Directeur de recherche, CNRS, Examinateur
M. David Goudin
Ingénieur Chercheur, CEA, Examinateur
M^me^ Pascale Rossé-Laurent
Architecte Logiciel, Bull, Examinatrice
M. Jean-François Méhaut
Professeur, Université Joseph Fourier, Directeur de thèse
M. Thierry Deutsch
Chercheur, CEA-Grenoble, Directeur de thèse

Remerciements

Après un peu plus de quatre années, voici venue l'heure des remerciements, qui scellent, avec un pincement au cœur, la fin de cette aventure.

Je voudrais commencer par remercier M. Édouard Audit et M. Pierre Manneback pour leurs rapports détaillés. Merci à M. Jean Virieux pour m'avoir fait l'honneur de présider le jury. Mes remerciements vont également aux autres membres du jury pour avoir accepté d'être présent durant ce moment si particulier qu'est la soutenance.

L'environnement scientifique et technologique qui a accompagné le déroulement de ces travaux de thèse était exceptionnel. Je remercie pour cela Thierry Deutsch, co-directeur de cette thèse, pour m'avoir fait profiter de toutes les facilités offertes par le CEA – notamment l'accès a certains supercalculateurs parmi les plus puissants au monde. Je n'oublie pas Luigi Genovese, cela a été un véritable plaisir de travailler sur BigDFT avec lui.

Je remercie la société Bull, pour avoir accepté de financer ma thèse. Merci à Pascale Rossé-Laurent, co-encadrante, pour sa vision industrielle des problématiques traitées dans cette thèse, et, plus généralement, pour m'avoir fait partager sa passion des défis technologiques qu'entraîne la conception de supercalculateurs toujours plus puissants.

Je voudrais vivement remercier Jean-François Méhaut, qui, en plus de la direction de ces travaux de thèse, m'a véritablement appris à rédiger des documents et des présentations scientifiques. Je réalise maintenant ô combien cela a du être fastidieux de m'accompagner durant ces longs mois qu'a nécessité l'écriture de ce manuscrit. Merci également pour toutes ces missions à l'étranger et ces rencontres qui m'ont apporté un extraordinaire enrichissement et qui ont énormément contribué à l'achèvement de ces travaux.

Je remercie également les membres des équipes avec lesquelles j'ai travaillé. Celles de Bull, notamment l'équipe kernel, les "bencheurs", et bien sûr, ma nouvelle équipe. Merci également aux membres du laboratoire L_SIM du CEA et de l'équipe MESCAL du LIG, notamment Augustin et Jérôme, qui ont participé à l'aventure S_GPU [1|2]. Merci à Dimitri Komatitsch pour m'avoir donné l'opportunité de travailler sur SPECFEM3D, et merci à Pieyre, pour cette formidable collaboration sur, justement, SPECFEM3D.

Mes derniers remerciements vont à Mélanie, sans qui ce document aurait certainement contenu un nombre incalculable de fautes d'orthographe.

Table des matières

Table des figures

Listings

Liste des symboles

GPU Graphics Processing Unit 17

FPGA Field-programmable gate array 18

DFT Density Functional Theory, ou *Théorie de la fonctionnelle de la densité* 9

CUDA Compute Unified Device Architecture 33

CPU Central processing unit 9

SIMD Single instruction, multiple data 11

MIMD Multiple instruction, multiple data 11

UMA Uniform memory access 12

NUMA Non-uniform memory access 12

FSB Front-Side Bus 12

QPI QuickPath Interconnect 13

DMA Direct memory access 15

RDMA Remote direct memory access 15

SM Streaming multiprocessor 19

SP Streaming Processor 19

MPI Message Passing Interface 26

API Application Programming Interface 26

Chapitre Premier

Introduction

La simulation numérique est devenue aujourd'hui incontournable pour quasiment la totalité des disciplines scientifiques, technologiques et industrielles. Un nombre considérable d'articles scientifiques emploie des méthodes basées sur la simulation pour valider des résultats ; il n'existe plus un projet industriel d'envergure où la simulation n'a pas été utilisée lors du processus de conception. Les chercheurs en physique peuvent ainsi simuler des phénomènes pour lesquels il aurait été difficile et coûteux de faire une expérience, comme la manipulation d'objets infiniment petits. Des industriels, comme les avionneurs, peuvent concevoir un avion uniquement par la simulation et éviter le plus possible des tests sur des maquettes réelles – ce qui réduit considérablement les coûts et les délais de conception –. Les avions récents de Dassault Aviation sont, par exemple, conçus par la simulation.

Le besoin en puissance de calcul de ces simulations est considérable ; plus la puissance de calcul disponible est importante, plus les modèles physiques peuvent être précis, ou plus la taille des objets simulés peut être augmentée. Cette course à la puissance demande des innovations matérielles continuelles afin de concevoir des supercalculateurs de plus en plus puissants. Aujourd'hui, cette puissance est produite à travers un parallélisme toujours plus massif. Cependant, un logiciel de simulation est généralement un logiciel de grande taille, sur lequel de nombreux contributeurs différents ont travaillé. Bâtir un tel logiciel peut nécessiter de gros moyens et de longues années. Il est dès lors très difficile pour les programmeurs de repenser leurs applications pour accompagner le développement des architectures des supercalculateurs.

Il est cependant clair que la notion de parallélisme – c'est-à-dire l'exécution simultanée de plusieurs instructions – est centrale lorsque l'on parle de calcul intensif. Les supercalculateurs modernes sont tous bâtis sous la forme de grappe de nœuds de calcul, et un gros effort a été fourni par les développeurs pour adapter leurs applications de simulation à ce type d'architectures parallèles. Mais la tendance actuelle est de créer de plus en plus de parallélisme à tous les niveaux de ces grappes de calcul. Ainsi, autrefois épargné par cette tendance, le processeur principal de la machine lui même devient parallèle. Actuellement il est courant d'avoir des processeurs de huit à dix cœurs possédant des instructions vectorielle pour chaque cœur. Un parallélisme de plus en plus massif est donc nécessaire pour exploiter les processeurs modernes.

Des problématiques de dissipation thermique et de consommation énergétique ont en effet stoppé l'augmentation continue de la fréquence de fonctionnement de ces processeurs. La tendance aujourd'hui est de limiter la fréquence et d'augmenter le nombre de cœurs au sein des processeurs. Ainsi, le paradigme du multicœur est une piste sérieuse pour relever un des grands défis qui se pose aux architectures du futur : la consommation énergétique.

On cherche ainsi à multiplier les cœurs d'exécution au sein des ressources de calcul.

Dans ce contexte, un parallélisme poussé à l'extrême a été appliqué à d'autres types de processeurs. Les accélérateurs graphiques, par exemple, sont des processeurs massivement parallèles constitués de plusieurs centaines de cœurs d'exécution très simples, qui sont capables de traiter très rapidement certains types de calcul. Bien que ces accélérateurs graphiques soient initialement destinés aux jeux vidéos, ils sont devenus tellement génériques que leur usage a été détourné pour accélérer des calculs généralistes. A titre d'exemple, les accélérateurs NVIDIA, de la génération KEPLER sont capables de traiter 10^{12} opérations flottantes double précision en une seconde (1 TFlops). La puissance du KEPLER est bien supérieure à celle d'un processeur généraliste récent de type Intel Xeon Sandy Bridge équipé de huit cœurs, qui peut soutenir, lui, environ 200 GFlops (200×10^9 opérations flottantes double précision par seconde).

Les machines hébergeant des accélérateurs sont donc constituées de plusieurs types de processeurs : un certain nombre de processeurs centraux et un certain nombre d'accélérateurs : les architectures de calcul deviennent donc hybrides. Nous nous sommes intéressés dans cette thèse à des problématiques liées à cette hétérogénéité des ressources de calcul au sein des grappes.

Problématiques et contributions

Problématiques

Bien que l'architecture matérielle des machines hybrides soit plus complexe à exploiter que les architectures homogènes traditionnelles, l'énorme potentiel des accélérateurs – et donc des architectures hybrides – intéresse vivement le monde du calcul haute performance. Nous nous sommes intéressés dans cette thèse à deux problématiques principales concernant l'exploitation des machines hybrides.

Partage des accélérateurs La première concerne le partage des accélérateurs pour les applications MPI. Il est en effet courant qu'il y ait un grand nombre de tâches MPI et peu d'accélérateurs dans un nœud hybride ; cela pose des problèmes de performance et également de répartition de charge. Les ressources de calcul deviennent en effet hétérogènes ; nous avons travaillé sur l'impact de cette hétérogénéité sur les applications et comment la limiter.

Programmation et exécution de code sur architecture hybride Nous nous sommes intéressés également à l'exécution concurrente de code entre CPU et accélérateur. Ceci pour exploiter en totalité la puissance d'une grappe hybride.

Ce type d'exécution concurrente nécessite un support au niveau langage de programmation afin d'exprimer qu'une exécution puisse être hybride. Nous visons des logiciels de simulation comportant plusieurs centaines de milliers de lignes de code ; nous avons donc chercher à comprendre comment minimiser l'intrusivité du support des accélérateurs dans ces application parallélisées uniquement pour des architectures parallèles homogènes.

De plus, une exécution hybride mettant en œuvre différents types de processeurs peut rapidement devenir complexe à appréhender. La compréhension d'une exécution et la dé-

tection d'erreur de programmation peuvent dès lors devenir difficiles mais sont nécessaires pour les programmer efficacement.

Contributions

Notre travail de thèse s'est concrétisé autour de quatre contributions.

- **Modèles de programmation** Tout d'abord, nous avons cherché à étendre les modèles de programmation employés dans les applications scientifiques pour traiter les problèmes de partage et d'exécutions concurrentes.
- **Déploiement hybride** La seconde contribution concerne le déploiement efficace du code sur ces architectures hybrides. En effet, le code défini à l'aide de nos modèles de programmation doit être ensuite exécuté sur une architecture hybride. L'ajout des accélérateurs complexifie le déploiement du code sur ces architectures. Il faut, par exemple, décider comment le code sera ordonnancé sur les accélérateurs, comment gérer une éventuelle parallélisation des traitements sur le processeur central et l'accélérateur ou comment optimiser les mouvements de données.
- **Analyse et compréhension** Les applications scientifiques s'exécutant sur des architectures hybrides doivent pouvoir être également appréhendées. Donner la possibilité au programmeur d'observer une exécution de son application est ainsi un moyen qui lui permet de comprendre son exécution. C'est notre troisième contribution
- **Validation** Nous avons ensuite validé nos propositions sur deux applications scientifiques : BigDFT et SPECFEM3D. Les différents développements logiciels effectués lors de cette thèse sont disponibles sous la forme de deux bibliothèques logicielles que nous avons appelées S_GPU 1 et S_GPU 2.

Contexte scientifique et industriel

Les travaux de cette thèse s'inscrivent dans le cadre d'une collaboration entre l'équipe MESCAL du Laboratoire d'Informatique de Grenoble (LIG), la société Bull et le laboratoire L_SIM de l'institut INAC au CEA Grenoble.

Le projet MESCAL du LIG a pour objectif de développer et d'étudier des grands systèmes (grappes et grilles de calcul) et d'en analyser les performances pour permettre un passage à l'échelle. Un des domaines d'application privilégié du projet MESCAL est celui du calcul intensif sur les grappes de calcul. Les problématiques informatiques de cette thèse ont été traitées avec MESCAL.

Bull est une société informatique présente sur plusieurs marchés comme le stockage, le service ou la sécurité informatique. Elle est également connue pour ses solutions pour le calcul haute performance. Elle a, par exemple, conçu le supercualculateur pétaflopique TERA100 pour le CEA et vend des supercalculateurs à des sociétés comme Météo France. La majorité des machines utilisées dans cette thèse a été mise à disposition par Bull.

L'INAC, ou Institut nanosciences et cryogénie, est un institut de recherche du CEA qui se distingue dans des domaines tels que les nanomatériaux, la photonique, la spintronique, le nanomagnétisme, la supraconductivité, la nanoélectronique ou encore les lésions de l'ADN. L'INAC est fortement impliqué dans la cryogénie des programmes spatiaux avec

ESA et des grands équipements comme ITER et le Laser Megajoule. Le CEA a apporté à cette thèse son expertise des applications scientifiques, en particulier BigDFT.

Organisation du document

Ce document est organisé en trois grandes parties. Chacune d'entre elles contient plusieurs chapitres. La première partie positionne les travaux de thèse. Nous verrons notamment dans le premier chapitre les différentes évolutions des architectures matérielles pour les applications scientifiques, des premières machines aux machines hybrides avec accélérateur. Le second chapitre présentera les environnements logiciels actuels pour exploiter ces architectures.

La seconde partie décrit les contributions de nos travaux de thèse. Tandis que le premier chapitre introduit les grandes idées qui ont conduit nos travaux, le second chapitre décrira la mise en œuvre des propositions à travers la description de nos deux bibliothèques logicielles : S_GPU 1 et S_GPU 2.

La troisième partie est consacrée à l'évaluation de nos propositions. Après avoir présenté, dans le premier chapitre, deux applications scientifiques sur lesquelles nos évaluations seront menées – BigDFT et SPECFEM3D –, nous présentons dans les deux chapitres suivants une évaluation de S_GPU 1 et S_GPU 2 sur ces deux applications scientifiques.

Première partie

État de l'art

Chapitre Deux

Architectures de calcul haute performance pour les simulations scientifiques

Sommaire

Le domaine du calcul haute performance est étroitement lié à celui des applications de simulations scientifiques ; celles-ci cherchent en effet continuellement à exploiter la grande puissance de calcul des architectures matérielles. Ce chapitre propose un état de l'art des architectures matérielles destinées au calcul scientifique, depuis les premiers calculateurs vectoriels jusqu'aux grandes grappes hybrides.

2.1 Les simulations scientifiques

Depuis l'apparition des premiers calculateurs, les scientifiques ont profité de leur puissance de calcul afin de les utiliser pour résoudre numériquement les équations de modèles physiques complexes. C'est encore plus vrai aujourd'hui où la puissance des nouveaux calculateurs permet d'appréhender la réalité avec une précision toujours plus grande.

Il va s'agir dans cette partie, de rappeler le contexte des simulations scientifiques, lesquelles constitueront les applications des contributions de cette thèse.

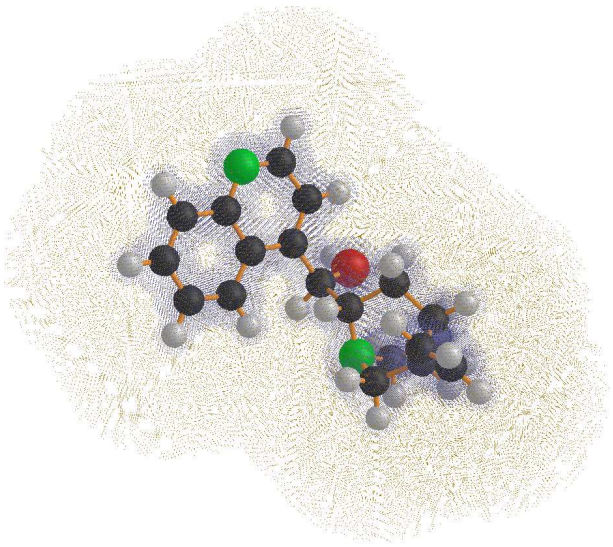


Figure 2.1 – Une molécule calculable par une simulation atomistique

2.1.1 Pourquoi simuler ?

De plus en plus de domaines, scientifiques ou industriels, emploient des simulations. Les raisons en sont multiples. Ces simulations rendent possible la visualisation de phénomènes difficiles, ou impossibles à appréhender par les expériences "classiques" du fait de leur taille ou de leur durée d'exécution. Par exemple, à grande échelle, il est possible de rejouer un tremblement de terre de nombreuses fois, en simulant le parcours des ondes sismiques dans la croûte terrestre, afin de saisir de manière détaillée le déroulement des évènements. A l'autre extrémité, à l'échelle de l'infiniment petit, la simulation permet d'isoler une unique molécule de quelques atomes (comme sur la figure 2.1) pour l'étudier précisément. Les logiciels de nanosimulation permettent de modifier la configuration d'une molécule pour en étudier la stabilité ou d'en calculer les propriétés physiques comme l'absorption ou l'émission de lumière.

Comme la physique, les acteurs industriels utilisent allégrement des simulations. L'aéronautique les exploite pour mettre au point des éléments aérodynamiques sans utiliser de coûteuses maquettes grâce à des modèles dynamiques d'écoulement de fluides. Ainsi, ces simulations permettent de faire des expériences à coût financier réduit, et dans le même temps réduisent le temps de conception des différents objets.

2.1.2 Modèles numériques des simulations scientifiques

La résolution des équations mathématiques issues d'un modèle physique requiert de choisir entre différentes méthodes numériques.

Une méthode bien connue est celle des éléments finis, qui consiste à résoudre numériquement des équations différentielles en maillant l'espace de façon plus ou moins complexe pour le discrétiser. Deux étapes sont donc nécessaires : le maillage de l'espace, effectué grâce à des logiciels spécialisés comme Gambit, Scotch ou Gmsh et la simulation proprement dite, qui se base sur ce maillage. Le logiciel de simulation SPECFEM3D [TKL08], par

exemple, utilise cette approche. D'autres domaines, comme la dynamique moléculaire, avec le logiciel NAMD [PBW+05], utilisent les lois classiques de la mécanique newtonnienne pour calculer, entre autre, la position et la vitesse des particules d'un milieu. D'autres simulations encore, se basent sur des théories très sophistiquées, comme la DFT [1] (basée sur la mécanique quantique), pour comprendre l'évolution de très petites structures atomiques : c'est le cas du programme BigDFT [GNG+08]. Les simulations basées sur cette théorie ont la propriété d'être prédictives car elles ne nécessitent pas de paramètres ajustables. D'autres méthodes sont employées comme la méthode de Monte-Carlo, [JB94] basée sur des procédés probabilistes.

Les résultats ainsi fournis doivent être confrontés à la réalité afin de contrôler la conformité de la simulation par rapport au phénomène physique simulé.

2.1.3 Un besoin en puissance de calcul

La grande majorité des simulations scientifiques a pour point commun de nécessiter une grand puissance de calcul. En effet, les modèles numériques, comme ceux présentés dans la partie précédente, nécessitent un grand nombre d'opérations mathématiques avant de converger vers une solution acceptable. Par exemple, pour les applications basées sur la DFT comme BigDFT [GNG+08], le besoin en calcul et en mémoire croît comme le cube de la taille du système. Pour un système de 1 500 atomes, il faut 3 000 Go de mémoire et une journée de calcul sur 1 500 cœurs CPU [2] d'une grappe de calcul moderne pour une optimisation des positions atomiques. Cette parallélisation est nécessaire pour diminuer le temps d'exécution mais aussi pour des raisons de mémoire : s'il y a 24 Go octet de mémoire disponible sur un nœud d'une grappe, il faudra nécessairement 100 nœuds pour une application utilisant 2 400 Go de mémoire.

Cependant, le temps et la mémoire nécessaires à l'exécution de simulations scientifiques peuvent être modulés par différents paramètres. Il est, entre autre, possible de jouer sur la précision des calculs : un maillage large rendra un calcul rapide à défaut d'être précis. Une diminution du nombre d'objets à simuler réduira également le temps total de la simulation.

Pour éviter de faire des compromis sur les résultats des simulations, le moyen idéal est d'augmenter la puissance de calcul et la capacité de mémoire disponible. Ainsi, les logiciels de simulations scientifiques ont toujours cherché à exploiter le mieux possible les architectures matérielles existantes, des architectures qui ont fortement évolué depuis les premières machines de calcul.

Une métrique communément utilisée pour comparer la puissance des machines de calcul est le flops, ou le nombre d'opérations flottantes par seconde. Nous utiliserons cette métrique dans toute la suite de ce chapitre lors de la description de l'évolution des différentes architectures de calcul.

2.2 Les machines vectorielles

Bien que la naissance du calcul informatique se confonde avec celle des premiers ordinateurs dans les année 50, on peut dater l'émergence du calcul scientifique moderne à

1. Density Functional Theory, ou *Théorie de la fonctionnelle de la densité*
2. Central processing unit

l'année 1975 avec le supercalculateur CRAY 1 [CR75], un processeur vectoriel mis au point par Cray Research. Un exemple du fonctionnement de ce processeur est présenté en 2.2.2.

Le concept d'ordinateur vectoriel, qui exploite le parallélisme de données naturel de certains types de calculs (par exemple le traitement d'un tableau) est un élément essentiel dans le monde du calcul haute performance. En effet depuis quelques années des éléments vectoriels font leur apparition dans les processeurs. Des unités de calcul vectorielles (comme SSE ou AVX) sont désormais disponibles dans n'importe quel processeur grand public. Les accélérateurs font également un usage important de concepts venus du monde du calcul vectoriel.

2.2.1 Modèle d'exécution synchrone

Un processeur vectoriel offre la possibilité d'exprimer qu'une même instruction puisse être appliquée à plusieurs données. Pour cela, un modèle d'exécution totalement synchrone est proposé : le processeur exécute une suite d'instructions vectorielles. Dans ce type de machines, les registres ne sont plus *scalaires* mais *vecteurs*. Ainsi, alors que sur une machine scalaire classique, une instruction d'addition prend deux nombres et renvoie la somme des deux, dans le monde vectoriel, une addition prend deux vecteurs de n éléments et produit un vecteur de n résultats. La valeur de n dépend de l'architecture considérée.

2.2.2 Évolution des architectures vectorielles

Le premier processeur vectoriel largement utilisé a été le CRAY 1 [CR75], de Cray Research en 1975. Son architecture piplinée permet d'obtenir une puissance de calcul relativement impressionnante pour l'époque, de l'ordre de 100 MFlops. (entre 100 MFlops et 130 MFlops sur un produit de matrice décrit dans [CR75]). Le CRAY 1 possède huit registres scalaires et huit registres vectoriels de 64 éléments. Sa mémoire est constituée d'un million de mots de 64 bits. La figure 2.2 montre comment l'architecture piplinée du CRAY 1 est utilisée pour calculer une addition de deux registres vectoriels. Sur cet exemple, on réalise l'opération $V1 = V1 + V2$, où $V1$ et $V2$ sont deux registres vectoriels de 64 éléments. On note VX_n la nième composante du vecteur VX. Une addition est produite en huit cycles : deux de transit (en gris sur la figure) et six cycles de calcul. L'unité de calcul permet à six additions d'être traitées ensemble dans le pipeline (pipeline à 6 étages). A partir du huitième top d'horloge, une addition est produite à chaque cycle.

Cray Research a continué sur le chemin des architectures vectorielles en explorant la voie des multiprocesseurs avec le CRAY X-MP[CR82] et le CRAY 2[CR85] avec une performance maximale proche du Gflops. D'autres fabricants ont continué sur la voie du vectoriel mais l'approche purement vectorielle des processeurs précédente pose des problèmes pour des algorithmes qui s'expriment difficilement de manière vectorisée.

Instructions vectorielles intégrées aux processeurs Aussi, de nombreux fabricants de processeurs, comme Intel ou AMD, ont intégré à des processeurs scalaires classiques des fonctionnalités vectorielles afin d'accélérer des calculs vectorisables. Ce sont les instructions MMX, SSE, 3dnow ! ou AVX. Il s'agit donc ici de processeurs globalement scalaires qui peuvent traiter des instructions vectorielles pour accélérer des calculs adaptés. La tendance va vers une augmentation de la largeur des unités vectorielles intégrées aux processeurs.

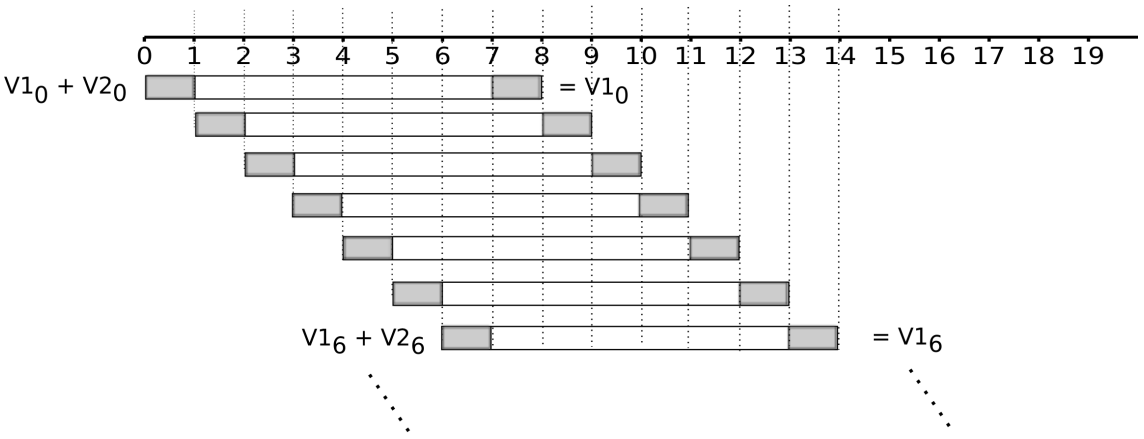


Figure 2.2 – Exécution piplinée du CRAY 1

Ainsi, tandis que les premières instructions SSE se contentaient de registres 128 bits (4 flottants simple précision), les unités vectorielles des processeurs Xeon Phi peuvent traiter des vecteurs de 512 bits (16 flottants simple précision ou 8 flottants double précision). Utiliser efficacement ces unités vectorielles devient donc de plus en plus indispensable pour exploiter la totalité des performances des processeurs modernes.

Il faut toutefois noter que des architectures purement vectorielles sont encore utilisées de nos jours, principalement dans le domaine météorologique. Ainsi, le supercalculateur japonais Earth Simulator, basé sur un ensemble de processeurs vectoriels NEC SX-6 a trôné au sommet du TOP500 de 2002 à 2004. Cependant, la tendance qui se dessine depuis deux décennies est la mise en commun de centaines de milliers de processeurs généralistes pour le calcul haute performance.

2.3 Les machines massivement parallèles

Les machines massivement parallèles peuvent se définir comme étant constituées d'un très grand nombre de processeurs interconnectés au sein d'un réseau rapide.

2.3.1 Modèles d'exécution

Les machines massivement parallèles proposent différents modèles d'exécution. Si la machine est SIMD [3] [Fly72], elle offre un modèle d'exécution proche des architectures vectorielles, donc totalement synchrone. En revanche, si l'architecture est MIMD [4] [Fly72], l'exécution devient asynchrone et des instructions différentes peuvent être exécutées simultanément.

3. Single instruction, multiple data
4. Multiple instruction, multiple data

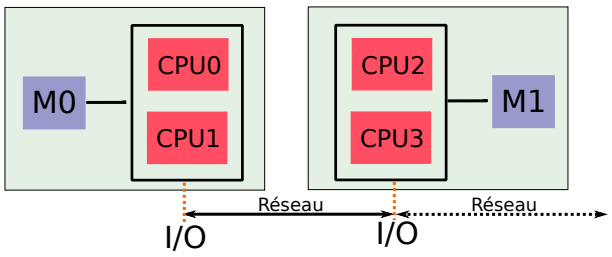


Figure 2.3 – Architecture à mémoire distribuée

2.3.1.1 Mémoire distribuée

Dans une architecture à mémoire distribuée, représentée sur la figure 2.3, chaque groupe de processeurs possède une mémoire. Ainsi, lorsque différents processeurs souhaitent partager des informations, ils passent par un réseau d'interconnexion afin d'échanger des messages. Ce type d'architecture permet d'être étendu d'une manière relativement simple en connectant à un nœud processeurs + mémoire un autre nœud du même type.

Il existe différentes topologies de réseaux d'interconnexion entre les nœuds d'une machine distribuée, comme la topologie en anneau ou la topologie fat-tree [Lei85a].

C'est ce type de machine qui est largement utilisé aujourd'hui. Il n'est pas rare d'avoir ainsi plusieurs milliers de nœuds connectés en réseau. A titre d'exemple, le calculateur TERA100 du CEA contient 4300 serveurs pour un total de 300 To de mémoire.

Pour éviter de multiplier les nœuds et les interconnexions, on essaye aujourd'hui d'augmenter la puissance des nœuds de calcul. Un moyen est par exemple d'exploiter des gros nœuds de calcul où les processeurs accèdent à une zone de mémoire partagée.

2.3.1.2 Mémoire partagée

Lorsque l'architecture matérielle est à mémoire partagée, différents processeurs sont directement connectés à une même zone mémoire qu'ils partagent entre eux. Il existe deux grandes familles d'architectures à mémoire partagée : les systèmes UMA [5] et NUMA [6].

UMA Dans une machine UMA (Figure 2.4), il n'existe aucune hiérarchisation de l'accès à la mémoire, et par conséquent, le temps d'accès y est identique pour chaque processeur du système. Dans ces architectures, la mémoire est donc une ressource globale partagée, chaque processeur utilisant le même canal de communication pour y accéder. Depuis le début des années 90, la technologie majoritairement employée pour réaliser ce canal de communication était l'utilisation du bus FSB [7], qui connectait les processeurs aux contrôleurs mémoire de la machine. Sur un système multiprocesseur, le bus FSB devait donc être partagé entre tous les processeurs : il devenait un goulot d'étranglement empêchant de multiplier les processeurs de la machine. Pour limiter l'effet de ce que l'on appelle le *memory wall*, on cherche à partitionner la mémoire du système.

5. Uniform memory access
6. Non-uniform memory access
7. Front-Side Bus

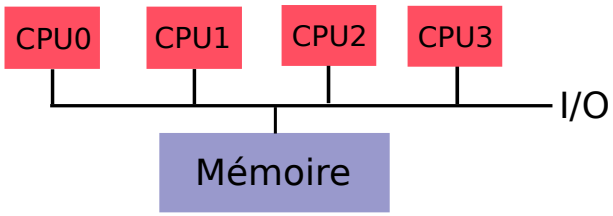


Figure 2.4 – Architecture UMA

NUMA L'architecture NUMA (Figure 2.5) a pour principal objectif de limiter le problème de congestion apparu dans les architectures UMA. En effet, chaque processeur a accès à toute la mémoire disponible mais celle-ci est découpée en zones. Ainsi, plutôt que de partager la bande passante d'un même lien d'accès à une zone mémoire, chaque processeur a un accès direct à une partition de l'ensemble de la mémoire. Ces architectures sont donc adaptées à la constitution de nœuds pouvant contenir un nombre très important de cœurs.

La figure 2.5 représente une machine NUMA constituée de quatre nœuds NUMA. Chaque nœud possède deux processeurs et une zone mémoire. Si un nœud désire faire une entrée-sortie ou accéder à la zone mémoire de l'autre nœud, il est contraint d'utiliser le réseau d'interconnexion représentée sur la figure par les lignes pointillées. De plus, si le nœud connecté à la mémoire M0 souhaite utiliser la mémoire M3, il doit passer par un autre nœud car il n'y a pas de liens direct entre ces deux espaces. Il y a donc une notion de hiérarchisation des accès à prendre en compte : lorsqu'un processeur accède à sa zone mémoire, son temps d'accès y est optimal. Lors d'un accès à une zone confiée à un autre processeur, ou accès distant, le temps d'accès se dégrade. Le rapport entre le temps d'accès local et distant est appelé facteur NUMA. Ce facteur dépend d'un grand nombre d'éléments comme le type de processeur, la nature des liens, ou les performances des mémoires. Cette hiérarchisation nécessite donc un placement rigoureux des calculs et de la mémoire. Il est donc préférable qu'un processeur utilise une zone de mémoire "proche" de lui. De même, lors de l'utilisation de bus d'entrée sortie, une mémoire "proche" devrait être utilisée pour, là encore, augmenter les débits des transferts.

Ce type d'architecture est très utilisée dans les grappes actuelles car elle permettent d'avoir des nœuds utilisant un grand nombre de processeurs. Les liens sont généralement de type Hypertransport [AMD09] d'AMD ou QPI[8], d'Intel. Ces liens fonctionnent en point-à-point et assurent une cohérence de cache globale à tous les niveaux de la hiérarchie mémoire. Pour prendre un exemple, la bande passante d'un lien QPI est de 25,6 Go/s, qui est le double de celle offerte par les dernières révisions du bus FSB.

Ces bonnes caractéristiques rendent ces deux types de réseau interne prépondérants aussi bien dans les nouvelles grandes grappes de calcul que dans les petits ordinateurs personnels.

8. QuickPath Interconnect

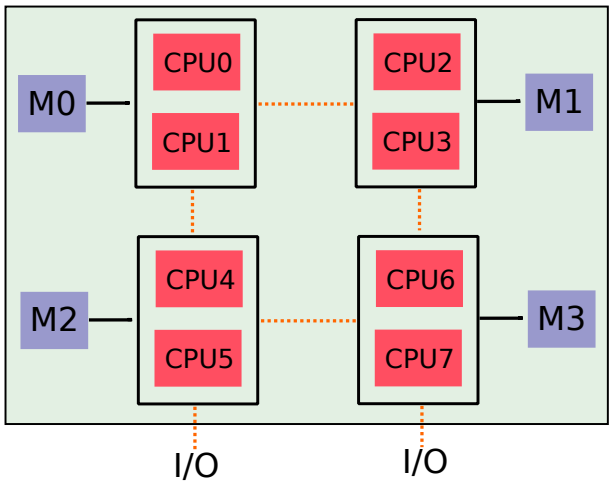


Figure 2.5 – Architecture NUMA

2.3.2 Les entrées sorties

Nous avons, dans la partie précédente, présenté les réseaux internes aux architectures, permettant aux processeurs de dialoguer avec la mémoire. Nous présentons ici les moyens qu'ont ces mêmes processeurs pour dialoguer avec l'extérieur via des connecteurs d'entrée sortie (ou IO [9]), c'est-à-dire avec des disque durs, des cartes réseau ou d'autres dispositifs comme les accélérateurs. Nous verrons également en quoi ces éléments sont importants lorsque l'on étudie les accélérateurs.

Bus d'entrée-sortie PCI Express Les réseaux internes aux machines (comme le QPI) ne sont pas exposés à l'extérieur du nœud. Des composants spéciaux, – appelés IOH [10] dans la terminologie Intel –, ont la charge d'interfacer le bus interne avec un bus externe standard d'entrée-sortie comme le bus PCI Express qui est le bus d'entrée sortie majoritairement utilisé aujourd'hui. PCI Express utilise une notion de liens ; chaque lien PCI Express (de génération 2) peut fournir 500 Mo/s de débit. Ainsi, si un dispositif en utilise 16, il pourra utiliser un débit maximum de 8 Go/s.

Dans le cas général, un bus interne ayant une bande passante limitée, il est incapable d'alimenter une infinité de lignes PCI express. Par exemple, un IOH de type *Tylersburg*, connecté sur un bus QPI ne peut fournir que 36 liens PCI Express. Si trop de dispositifs sont connectés, cette bande passante devra être partagée entre eux. Il est néanmoins possible de contourner ce problème en utilisant plusieurs IOH, en les connectant sur le bus QPI de plusieurs processeurs. Le bon dimensionnement des liens PCI Express reste d'actualité pour les processeurs plus récents, de type Sandy/Ivy Bridge, qui incorporent le contrôleur PCI Express.

La grande majorité des accélérateurs actuels (par exemple les GPU et les Xeon Phi) se connecte sur les bus PCI Express ; ces bus se comportent véritablement comme un goulot d'étranglement et toute architecture matérielle mise au point sérieusement évitera

9. Input Output
10. Input Output Hub

d'utiliser un nombre d'accélérateurs requérant plus de bande passante que celle disponible. Cela limite le nombre d'accélérateurs installés dans les machines.

Les cartes réseau La connexion entre un nœud et le réseau externe de la grappe se fait par l'intermédiaire des cartes réseaux. Celles-ci sont pratiquement de petits ordinateurs, possédant mémoire et processeurs, permettant de décharger le processeur central du nœud de la gestion des détails des protocoles réseaux. Ces cartes possèdent également des dispositif d'accès directs à la mémoire centrale (mécanisme DMA [11]). Ainsi, les copies mémoire entre un nœud et la carte réseau peuvent être effectuées indépendamment du processeur central. Par extension, les environnements réseaux sont capables d'effectuer des requêtes RDMA [12], où un nœud, par l'intermédiaire des mécanismes de DMA va pouvoir lire ou écrire directement dans l'espace mémoire d'un autre nœud sans passer ni par le processeur ni par le système d'exploitation. Avec des mécanismes adaptés de détection de la fin d'une opération [MGH+96], les transferts DMA permettent d'atteindre de très bonnes performances. Il est important de noter que les pages mémoires concernées par un transfert DMA doivent obligatoirement être "punaisées" (*pinned memory*) afin d'empêcher le mécanisme de *swapping* du système d'exploitation d'agir sur ces pages.

Les cartes réseau utilisent elles aussi un bus PCI Express. Les accélérateurs comme les GPU, que nous verrons en détail ultérieurement, sont également connectés au bus PCI Express. Le dimensionnement de ces bus est donc essentiel lors de la mise au point d'un nœud pour éviter d'avoir à partager une bande passante limitée.

Pour maximiser les performances, ces grappes utilisent principalement des réseaux dits à *haute performance*, qui cherchent à réduire la latence tout en maximisant le débit. Les deux principales technologies employées dans les machines de calcul haute performance actuelles, d'après le TOP500 de novembre 2012, sont Gigabit Ethernet [Sei98] et Inifiniband [IB00].

2.3.3 Les grappes de calcul multiprocesseur

Les plus puissants calculateurs actuels sont basés sur une architecture dite de grappe de calcul, où un ensemble de nœuds est connecté en réseau. Chaque nœud est une machine multiprocesseur à mémoire partagée (UMA ou NUMA).

Les différents nœuds de ces grappes sont, par l'intermédiaire de leurs cartes réseau, connectés entre eux suivant une topologie déterminée. La topologie *fat-tree* [Lei85b] est largement employée dans les grappes de calcul actuelles. Cette topologie consiste en un arbre où la bande passante n'est pas homogène sur tous les liens : plus on descend vers les feuilles, plus la bande passante entre les liens diminue. Les nœud n'étant pas connectés entre-eux directement, des commutateurs, permettant de faire circuler les messages entres différents nœuds, sont nécessaires.

Comme les machines massivement parallèles historiques, chaque nœud utilise des composants standards (mémoire, disque dur et processeur standard). Les interconnexions sont généralement basées sur Infiniband ou Gigabit Ethernet. Les machines les plus puissantes

11. Direct memory access
12. Remote direct memory access

du classement TOP500 de novembre 2012 sont toutes des grappes de calcul, comme “Titan” (classée 1re) ou le “K computer” (classée 3e).

2.3.4 Synthèse

Historiquement, depuis les premières machines massivement parallèles jusqu'aux grappes de calcul actuelles, les machines étaient homogènes : elles possédaient des processeurs identiques avec une quantité de mémoire par processeur également identique. Pour augmenter la puissance disponible de ces grappes, la question de maintenir une homogénéité de puissance est posée.

Nous arrivons en effet à un stade où la puissance consommée par ces calculateurs devient énorme (le K-computer de Fujitsu consomme une puissance de 12 GW). La question centrale est donc de déterminer comment augmenter la puissance tout en maintenant une consommation réduite. Augmenter la fréquence des processeurs est une mauvaise idée : Intel montre que dans ses processeurs [Ram06], lorsque l'on augmente la fréquence de 20%, la puissance de calcul augmente de 13% alors que la consommation augmente de 73%.

La voie qui est suivie actuellement est d'augmenter le nombre de processeurs dans les nœuds des grappes. La tendance actuelle va vers l'utilisation massive de processeurs peu puissants et donc peu consommateurs d'énergie. Plusieurs solutions sont envisageables pour mettre en place cette augmentation de processeurs.

Une approche est d'utiliser uniquement des processeurs faible consommation homogènes (par exemple ARM). Ce type de machine est encore en phase de recherche et pose de nombreuses questions, comme celle de la scalabilité : ce genre d'architecture consistera en un grand nombre de nœuds peu puissants et une infrastructure réseau efficace devra être proposée.

Une autre approche est de regrouper ces processeurs supplémentaires dans des composants spéciaux – les accélérateurs. Les nœuds de calcul de telles architectures sont donc hybrides puisqu'ils ont deux types d'unités de traitement : les cœurs généralistes et les accélérateurs. On passe donc d'une architecture de puissance homogène à une architecture de puissance hétérogène. Contrairement à l'architecture précédente, celle-ci peut être vue comme un ensemble réduit de nœuds puissants.

2.4 Les machines parallèles hybrides

Pour améliorer encore la puissance des machines parallèles, des recherches sont menées pour utiliser des co-processeurs spécialisés permettant d'accélérer certains types d'opérations gourmandes en calcul. Des machines parallèles associées à ces co-processeurs accélérateurs sont appelées machines hybrides.

2.4.1 Principaux types d'accélérateurs

Les accélérateurs des machines hybrides peuvent être de plusieurs types, mais ils ont tous la particularité d'être particulièrement puissants dans le traitement de certains types de calculs comme des calculs lourds et répétitifs sur une grande quantité de données. Leur

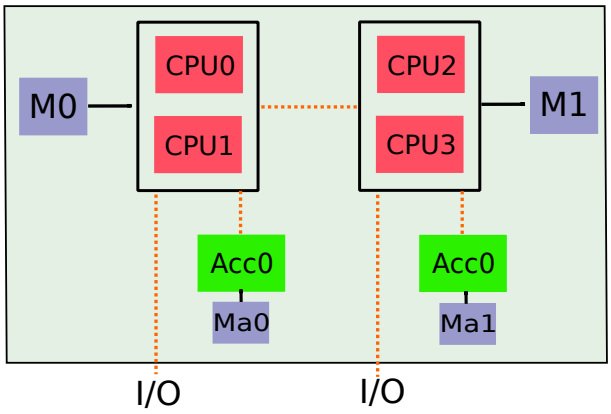


Figure 2.6 – Une machine hybride

puissance provient, pour une certaine partie, d'idées venues du monde vectoriel, où des traitements identiques sont effectués sur un grand nombre de données.

Une caractéristique d'architecture partagé par l'ensemble des accélérateurs – comme les GPU ou les Xeon Phi – consiste en la mise en commun d'un grand nombre de cœur d'exécution, relativement simple, qui permet de dégager une grande puissance de calcul lorsqu'ils sont utilisés ensemble. Nous verrons les détails d'architecture un peu plus loin mais l'on peut considérer ces accélérateurs comme de véritables petites machines parallèles encapsulées dans un circuit électronique. Ces caractéristiques (miniaturisation, nombre important de cœurs simples) assurent aux accélérateurs un ratio performance par Watt réduit.

Ces accélérateurs doivent être vus comme des coprocesseurs. Les processeurs centraux, appelés ici processeurs généralistes, vont contrôler ces accélérateurs : les accélérateurs ne sont donc pas autonomes. Même les Xeon Phi qui, ont une large autonomie, sont pour l'instant toujours dépendants d'un processeur généraliste. La figure 2.6 représente cette organisation où quatre processeurs généralistes contrôlent deux accélérateurs. Cet exemple, typique d'une installation à base de processeurs accélérateurs graphiques (GPU [13]) montre que le nombre de processeurs et le nombre d'accélérateurs n'ont aucune raison d'être identiques. Des accès concurrents de plusieurs processeurs à un accélérateur doivent donc être envisagés dans les réflexions consistant à établir le meilleur moyen d'utiliser ces accélérateurs.

De plus, une caractéristique remarquable de la plupart des accélérateurs actuels est leur mémoire dédiée. Les traitements sont effectués dans la mémoire de l'accélérateur et les processeurs généralistes sont responsables de la réalisation des transferts entre la mémoire centrale et la mémoire des accélérateurs. Selon la technologie de connexion employée (PCI express ou bus dédié), ces transferts peuvent être très coûteux, et l'obtention de bonnes performances dépend bien souvent des stratégies employées pour effectuer ces transferts.

13. Graphics Processing Unit

2.4.1.1 Les circuits électronique programmables

Les circuits logiques programmables, communément appelés FPGA [14] sont une classe de processeurs pour lesquels le matériel est configurable. La majorité des FPGA se compose de différents blocs inter connectés. Ces blocs peuvent être programmés pour effectuer des opérations mathématiques particulières. Un bus d'interconnexion, entièrement configurable lui aussi, permet de connecter ces différents blocs entre eux et ainsi de mettre au point un calcul spécifique.

Un calcul doit être ainsi programmé dans un langage de description de matériel comme VHDL [Ash01] ou Verilog [ver96]. Des logiciels spécialisés, comme ISE de Xilinx permettent de réaliser toutes les étapes de la conception (création des circuits, compilation et déploiement dans les FPGA).

Il existe une machine appelée Maxwell [BBB+08], composée de 32 nœuds comportant chacun deux FPGA Xilinx Virtex4, consacrée au calcul scientifique. Cependant, l'usage des FPGA est marginal dans le calcul scientifique car les langages de programmation destinés à créer des circuits (VERILOG, VHDL) sont très éloignés de ceux utilisés généralement dans le calcul scientifique (Fortran, C...).

2.4.1.2 Les accélérateurs graphiques

Les accélérateurs graphiques (appelés ici indifféremment (co)processeur graphique, ou GPU pour Graphic Processing Unit) sont une classe de coprocesseurs qui a récemment été introduite en production dans les machines de calcul haute performance.

À l'origine, ces processeurs étaient très peu programmables et difficilement utilisables en dehors du rendu graphique 3D pour lequel ils ont été conçu. Leur modèle d'exécution était en effet très proche du pipeline graphique de Direct3D ou d'OpenGL. Les évolutions successives de ces GPU ont entraîné une possibilité de programmation de plus en plus poussé de deux unités principales du pipeline graphique : les *vertex shaders*, qui s'occupent principalement de transformation de figures géométriques et les *pixel shaders* qui vont permettre d'appliquer des textures sur des objets 3D. La programmabilité de ces unités va entraîner plusieurs tentatives pour détourner le pipeline graphique du rendu d'image 3D vers du calcul généraliste [OLG+07]. Bien que la puissance de calcul produite soit tout à fait acceptable, il faudra attendre d'autres évolutions pour que le calcul généraliste sur GPU devienne réellement employé.

Avec l'arrivée de Direct3D 10 qui définit un seul type de *shader*, NVIDIA et ATI ont respectivement, dans leurs processeurs de type GeForce 8800 [LNOM08] et Radeon HD2000 [ATI11], unifié le vertex et le pixel shader. Pour la partie graphique, cela a permis entre autre de régler des problèmes d'équilibrage de charge (un vertex shader pouvant être plus chargé qu'un pixel shader lors de la création d'une scène 3D). Pour la partie calcul, cette unification a permis de simplifier davantage le calcul généraliste en visualisant les GPU comme des processeurs massivement multi cœurs. Cette étape a véritablement fait émerger les GPU comme une solution crédible de calcul haute performance. C'est d'ailleurs à ce moment que NVIDIA et ATI ont lancé leurs environnements de programmation généraliste pour GPU.

14. Field-programmable gate array

NVIDIA propose maintenant des cartes uniquement destinées aux calculs qui ne possèdent aucune sortie graphique.

Transferts mémoire Les cartes GPU sont toutes connectées sur le bus PCI Express et possèdent leur propre mémoire : des transferts de données doivent donc être effectués de la mémoire centrale vers la mémoire des GPU. Comme les cartes réseaux, les cartes GPU possèdent des unités DMA et c'est avec leur aide que les copies entre la mémoire de la carte et la mémoire centrale sont effectuées. Lors de communications entre les GPU et la mémoire centrale, des travaux [Hov08] ont mesuré des latences de l'ordre de $10\mu s$ et des débits de 6 Go/s. Des expériences analogues effectuées sur des machines Bull ont montré des résultats similaires. Cette latence élevée impose de privilégier de gros transferts plutôt qu'une multitude de petits.

Les transferts sont initiés par un processeur particulier, ainsi, dans le cas de système multiprocesseur, où certains processeurs sont éloignés du bus d'entrée-sortie sur lequel se trouve l'accélérateur, les transferts mémoire peuvent en souffrir. Des solutions logicielles permettant de placer au mieux les applications sur les processeurs ont donc été mises au point, comme le GPU-set [15] de Bull.

Les accélérateurs graphiques récents ont tous mis en place des mécanismes destinés à réduire l'impact des transferts mémoire, qui sont, comme vu, très coûteux. Il est par exemple possible d'initier un transfert pendant que la carte calcule afin de cacher le temps de ce transfert. Certaines cartes, comme les NVIDIA FERMI ou KEPLER, possèdent deux contrôleurs DMA et ont ainsi la possibilité de gérer simultanément deux transferts mémoire. Les applications GPU performantes utilisent ces possibilités pour optimiser les temps d'exécution des codes.

L'architecture de carte NVIDIA TESLA Nous décrivons ici l'architecture des cartes NVIDIA TESLA [LNOM08, NVI11a] qu'il ne faut pas confondre avec les cartes professionnelles TESLA sans sorties graphiques.

La figure 2.7 montre l'architecture d'une carte NVIDIA TESLA. Il est utilisé ici la terminologie NVIDIA pour décrire les objets constitutifs de la carte. Chaque SM [16] contient huit SP [17] qui gèrent les calculs entiers et flottants simple précision, deux unités permettant de traiter des fonctions "spéciales" (trigonométrie...) et une unité flottante double précision. L'unité d'ordonnancement de base d'un GPU NVIDIA est le WARP qui consiste en une opération vectorielle SIMD effectuant 32 opérations. Ainsi, lorsqu'un WARP effectue des opérations simple précision, il est exécuté en quatre coups d'horloge (un quart de WARP à chaque coup) par un SP. Comme un SP contient huit fois moins d'unités double précision que simple précision, les performances double précision sont considérablement inférieures à la simple précision sur les TESLA. Ainsi, un WARP contenant des opérations double précision sera exécuté en 32 coups d'horloge.

Les SM sont réunis dans des TPC (*texture processor cluster*). Il y a trois SM par TPC qui partagent une mémoire cache sur les opérations de texture. Chaque TPC est interconnecté et peut accéder à la mémoire globale de la carte. Le nombre de TPC, le

15. `www.bull.fr/extreme-computing/systeme_exploitation.html`
16. Streaming multiprocessor
17. Streaming Processor

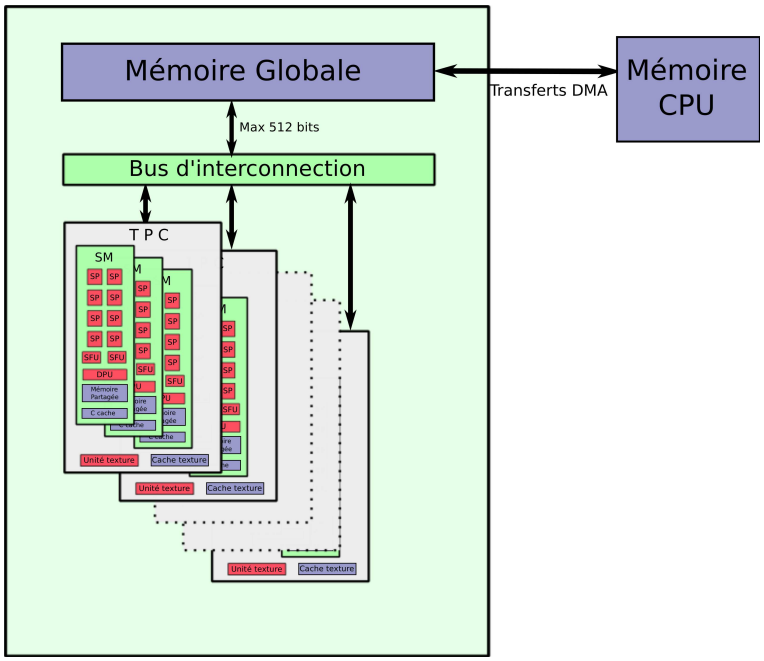


Figure 2.7 – Un accélérateur graphique TESLA connecté à une mémoire CPU

nombre de liens entre les TPC et la mémoire varient en fonction de la gamme des cartes TESLA. Cependant, les plus puissantes d'entre-elles possèdent dix TPC (pour un total de deux cent quarante SP), un bus mémoire de 512 bits et 4 Go de mémoire pour une puissance d'environ 900 Gflops en simple précision et 80 Gflops en double précision.

Le modèle d'exécution d'un programme sur une carte TESLA est appelé par NVIDIA SIMT (*Single Instruction, Multiple Thread*). La carte reçoit en effet des threads mais elle les exécute par groupe de 32 (les WARPs) sur une seule instruction SIMD. Ainsi, les performances sont maximales lorsque tous les threads d'un WARP exécutent la même instruction. Dans le cas contraire, l'exécution n'est plus parallèle mais sérialisée : il y a alors divergence du chemin d'exécution des threads.

L'architecture NVIDIA FERMI Les carte de génération TESLA souffraient d'un gros problème de performance en double précision (8 fois moins rapide que la simple). Malheureusement, un très grand nombre d'applications scientifiques – comme BigDFT – demande des calculs en double précision. Pour apporter une réponse à ce problème, NVIDIA a lancé l'architecture FERMI [NVI], qui a largement amélioré les performances des GPU pour le calcul généraliste. La carte reste SIMT et massivement multiprocesseurs, avec des caches ajoutés. Les performances en double précision deviennent très intéressantes (500 Gflops) et seulement deux fois moins puissantes que les calculs en simple précision (1 Tflops).

Un autre problème des cartes TESLA est l'absence de support de mémoire de type ECC, c'est-à-dire des mémoires qui contiennent des codes correcteurs d'erreurs pour rendre les calculs extrêmement fiables. Des applications critiques, médicales ou militaires par exemple, requièrent ce type de mémoire par sécurité. Ainsi, en plus de l'amélioration gé-

nérale des performances, l'architecture FERMI permet l'utilisation de ce type de mémoire.

L'architecture NVIDIA KEPLER Avec la dernière génération de GPU NVIDIA disponible, les KEPLER, les performances ont encore été augmentées. Il est possible d'atteindre 1 Tflops en double précision et 3 Tflops en simple.

Les accélérateurs graphiques AMD AMD produit également des cartes graphiques extrêmement puissantes pour le calcul. La carte Radeon HD 6970 [ATI11], architecturée autour de plusieurs centaines de processeurs SIMD permet d'atteindre une puissance simple précision de 2,7 Tflops et de près de 700 Gflops en double précision.

2.4.1.3 Les co-processeurs Intel Xeon Phi

Intel commercialise depuis début 2013 une toute nouvelle architecture de calcul appelée Xeon Phi. La génération actuelle de Xeon Phi se présente sous la forme de carte PCI Express dans laquelle les ingénieurs d'Intel ont intégré dans un seul processeur une soixantaine de cœurs d'exécution. Afin de générer une telle densité, une simplification des cœurs a été réalisée. Les cœurs des Xeon Phi sont ainsi beaucoup plus simples que ceux des Xeon actuels : par exemple, l'exécution du code sur les cœurs est in-order. Comme les cœurs ont deux unités d'exécution, Intel a introduit de l'hyperthreading dans ces cœurs ; les différentes unités d'exécution sont alimentées par des hyperthreads. Chaque cœur physique a un niveau d'hyperthreading de quatre ; ainsi pour un KNC comportant 57 cœurs physiques, le système en voit 228 (57x4). Il est donc indispensable d'utiliser ces hyperthreads (au moins deux par cœurs) pour exploiter toutes les possibilités des cœurs des Xeon Phi. Une application efficace sur Xeon Phi doit dès lors être extrêmement parallèle.

Chaque cœur des Xeon Phi possède une unité vectorielle, qui peut traiter en une instruction plusieurs opérations flottantes ou entières. Ces unités sont d'un type particulier ; contrairement aux unités vectorielles de type AVX disponibles sur les derniers Xeon, celles qui sont disponibles sur le Xeon Phi ont des registres deux fois plus gros (512 bits). Il est donc possible de traiter simultanément seize éléments en simple précision ou huit éléments en double précision.

Dès lors, pour exploiter correctement un Xeon Phi, les applications doivent être extrêmement parallèles et également extrêmement vectorisables. Bien que l'architecture soit très récente, un calculateur basé sur des Xeon Phi a déjà été classé 8^{e} dans le TOP500 de novembre 2012.

2.4.1.4 Un processeur hybride : Le Cell

Le Cell est un processeur développé dans le cadre d'une collaboration entre IBM, Sony et Toshiba. Il contient, sur un même circuit, deux types d'architecture de calcul. (figure 2.8). Il possède un Power Processor Element (PPE), basé sur une architecture classique PowerPC associée à huit Synergistic Processing Elements (SPE) basés sur une architecture vectorielle SIMD. Ces deux processeurs communiquent entre eux grâce à un bus d'interconnexion appelé Element Interconnect Bus (EIB). Comme les GPU, le Cell a des fonctions permettant de recouvrir un calcul avec un transfert mémoire. Le Cell

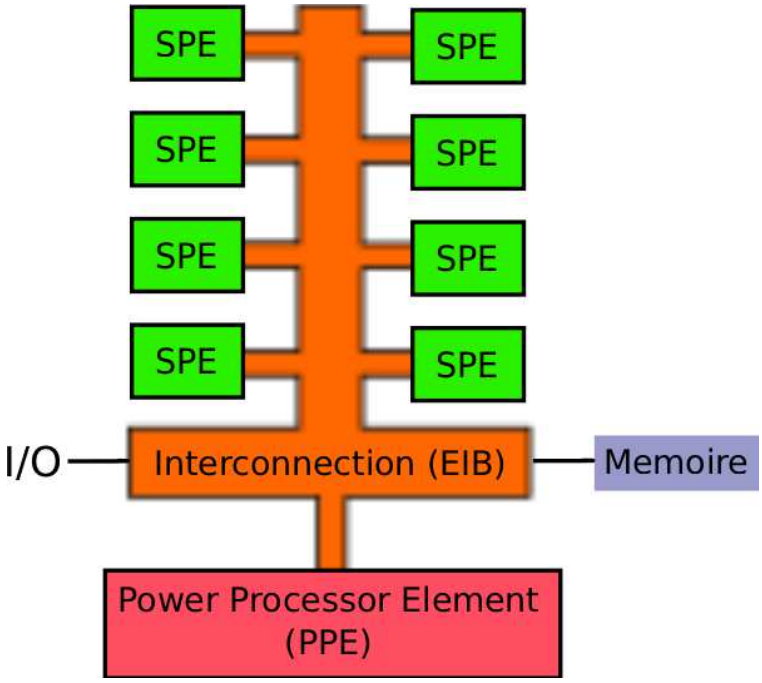


Figure 2.8 – L'architecture du processeur CELL

peut donc être vu comme l'association d'un processeur généraliste (le PPE) et de huit accélérateurs (les SPE). Le Cell est ainsi un processeur hybride.

Le Cell a démontré qu'il était possible de construire de puissants centres de calcul en utilisant des accélérateurs (le roadrunner fut 1er du TOP500). Cependant, chaque SPE s'occupant uniquement d'une très petite zone de mémoire, de nombreux mouvements de données doivent être mis en place, ce qui rend sa programmation très difficile. Nous verrons dans le chapitre suivant que les GPU ont des environnements puissants de programmation qui les rendent plus faciles à programmer : ils ont donc pris la succession du Cell dans les centres de calcul hybrides.

2.4.2 Les grappes de calculs hybrides

Des grappes de calculs dans lesquelles des accélérateurs ont été ajoutés dans les nœuds de calcul ont été construites : ce sont des grappes de calcul hybrides. La première grappe de calcul hybride véritablement destinée au calcul haute performance a permis d'atteindre pour la première fois le petaflop ; c'était le roadrunner en juin 2008, architecturé autour de processeurs CELL. Depuis, les machines à base de CELL sont en perte de vitesse, et de nouvelles grappes hybrides basées sur des GPU sortent et se classent régulièrement dans les premières places du classement TOP500. En novembre 2012, il y a trois machines hybrides dans les dix premières places du classement, deux machines GPU NVIDIA, une machine Xeon Phi. Par rapport à une grappe homogène, les grappes hybrides ajoutent deux particularités :

Mouvements de données L'ajout des accélérateurs dans les grappes de calcul demande de gérer des mouvements de données qui n'étaient pas nécessaires sur des grappes homogènes. Comme les zones de mémoire centrales et accélérateurs sont séparées, il faut en effet, en plus des transferts classiques de type nœud à nœud, initier des transferts CPU vers accélérateurs. Minimiser le coût de ces transferts est une conditions pour obtenir de bonnes performances.

Exécution simultanée CPU – GPU En plus de ces mouvements de données, on peut prendre en compte la possibilité d'utiliser simultanément les CPU et les accélérateurs dans les architectures hybrides : il est en effet possible de faire travailler simultanément les CPU et les accélérateurs. Utiliser ces deux ressources de calcul permet d'exploiter la grappe hybride en totalité mais complexifie grandement l'exploitation de ces architectures.

2.4.2.1 Un exemple de grappe hybride : Titane

Le premier grand calculateur français hybride, conçu par Bull en 2009, toujours en activité, s'appelle Titane. Il est scindé en deux parties. La première partie, conventionnelle, comporte 1 068 nœuds de calcul comptant chacun deux processeurs quadri-cœur Intel Nehalem associés à 24 Go de mémoire. Les nœuds sont connectés entre eux grâce à un réseau infiniband. C'est la seconde partie de Titane qui est hybride. Le supercalculateur est architecturé autour de 96 nœuds, identiques à ceux de la partie conventionnelle, connectés à deux cartes accélératrices NVIDIA TESLA.

Titane est typique des grappes de calcul hybrides actuelles : il possède des processeurs CPU puissants (de génération Intel Nehalem) associés à des accélérateurs GPU, et par conséquent un nombre élevé de processeurs partageant deux GPU (ici le ratio est quatre processeurs pour un GPU).

2.5 Conclusion

Les machines de calcul ont constamment évolué afin d'offrir la plus grande puissance de calcul possible aux applications scientifiques. La tendance, depuis une décennie est nettement en faveur des architectures massivement parallèles. Les technologies actuelles, à base de grappes de calcul (hybrides) permettent d'atteindre une puissance de quelques dizaines de petaflops.

Pour augmenter la puissance de calcul, la montée en fréquence des processeurs n'est plus possible ; la consommation énergétique devient en effet un problème critique et les processeurs multicœurs sont une piste sérieuse pour empêcher l'explosion de cette consommation. Nous assistons donc à une importante augmentation du degré de parallélisme des architectures parallèles existantes : les processeurs deviennent de plus en plus parallèles, et les accélérateurs qu'on y ajoute sont déjà des petites machines extrêmement parallèles.

Cette parallélisation massive n'est pas prête de s'arrêter. En effet, pour les prochaines générations de calculateurs, de classe exaflopique, on assistera très certainement à un accroissement considérable du nombre de ressources matérielles (CPU, disques, mémoires, accélérateurs...). Pour accompagner ces changements architecturaux, les applications scien-

tifiques devront être adaptées pour exploiter ce parallélisme massif. Dans le cas contraire, elles ne pourront exploiter qu'une infime partie des grappes de calculs.

Cependant, au vue des architectures matérielles considérées, la programmation d'architecture hybride peut être extrêmement délicate. Elle demande, en effet, au programmeur de prendre en compte différents niveaux de parallélisme et de transferts mémoire, de gérer un fonctionnement simultané d'accélérateurs et de processeurs et de prendre en compte un partage entre les CPU et les accélérateurs.

Ces problématiques demandent des environnements logiciels performants – dont l'intégration dans les applications existantes peut être plus ou moins intrusif – et une très bonne compréhension des différents niveaux de parallélisme pour leur programmation. Dans ce contexte, l'objectif du chapitre suivant est de présenter les différents environnements logiciels existants destinés à programmer de telles architectures parallèles.

Chapitre Trois

Programmation des grappes de calculs hybrides

Sommaire

Les modèles de programmation imaginés pour les grappes de calcul homogènes ne sont plus suffisants pour les grappes de calcul hybrides. En effet, avec l'ajout accélérateur, nous passons de grappe à nœuds multicœur à des grappes à nœuds multicœur hybrides. De nouveaux modèles de programmation sont donc à définir pour exploiter efficacement ces architectures hybrides.

Aussi, l'objectif de ce chapitre est de montrer comment ces architectures hybrides sont exploitées par les applications de simulation scientifiques. Pour cela, nous verrons tout d'abord les environnements de programmation classiques utilisés par les applications scientifiques dans un environnement homogène, suivi d'une description des environnements de programmations spécifiques aux accélérateurs. Nous montrerons ensuite comment ces deux types d'environnements peuvent être combinés pour utiliser les grappes hybrides. Nous finirons ce chapitre par la présentation d'outils destinés à comprendre le déroulement d'une exécution dans une grappe hybride.

3.1 Programmation des machines massivement parallèles

Il existe différents modèles de programmation adaptés aux différentes architectures matérielles des machines massivement parallèles. Certains sont plutôt destinés aux architectures à mémoire distribuée, d'autres aux architectures à mémoire partagée.

3.1.1 Passage de messages avec MPI

Le modèle de passage de messages est l'un des grands modèles parallèles existant. La parallélisation d'un problème est effectuée avec l'emploi d'un certain nombre de processus s'exécutant de manière concurrente. Chacun d'entre eux ayant uniquement accès à une zone de mémoire privée. Lorsqu'ils doivent partager des informations, ils échangent des messages. Ainsi, les processus n'ont aucun besoin de partager de la mémoire pour communiquer et par conséquent, la programmation par passage de messages est particulièrement adaptée aux machines où la mémoire est distribuée.

Il existe différentes mises en œuvre du paradigme à passage de message, comme PVM (Parallel Virtual Machine) [BDS96] ou MPI [mpi09, SOW+95]. Nous allons ici nous concentrer sur MPI, qui est la plus utilisée.

MPI MPI[1] [mpi09, SOW+95] est une norme qui a été mise au point dans le milieu des années 90. Elle définit une interface de programmation permettant de mettre en œuvre des communications point-à-point, collectives et des synchronisations entre différents processus. Cette interface peut être au choix synchrone ou asynchrone. Construire une application parallèle MPI consiste à écrire un programme unique qui sera dupliqué n fois. Il est attribué à chaque instance du programme un numéro d'identification unique (ou rang) afin de pouvoir sélectionner quels processus communiquent. Une notion de *communicateur* est définie permettant de créer des groupes de processus et ainsi de limiter la portée des messages aux membres d'un même groupe.

La figure 3.1 représente un programme MPI, écrit en C, où les processus de rang 0 et 1 communiquent. Le premier envoie un nombre entier (ligne 19) tandis que le second se met en attente pour le recevoir (ligne 23). Cet exemple utilise la version synchrone de l'interface de programmation (API[2]) MPI.

Beaucoup de mise en œuvre de la norme MPI Il existe beaucoup de mises en œuvre de la norme MPI. Habituellement, les constructeurs de machines parallèles livrent des versions spécialement optimisées pour leur matériel. La plupart se basent néanmoins sur des projets comme MPICH [MPI] ou OpenMPI [GFB+04]. Le matériel sous-jacent offre plusieurs types d'optimisations qui sont bien souvent utilisées par les bibliothèques MPI. Dans un environnement infiniband, les communications réseau utilisent une technique de RDMA (*remote direct memory access*), permettant à la carte infiniband d'utiliser directement la mémoire centrale sans copie supplémentaire. De plus, lorsque plusieurs processus peuvent partager de la mémoire, des mécanismes permettent d'échanger des messages sans copie.

1. Message Passing Interface
2. Application Programming Interface

D'autres mises en œuvre de MPI, comme AMPI pour *Adaptive Message Passing Interface* [CH03], basée sur la bibliothèque Charm++ [Kal90], vont au delà de la norme MPI et proposent, par exemple, des mécanismes évolués d'équilibrages de charge sur une grappe de calcul.

Un modèle de programmation largement utilisé par les grandes simulations scientifiques Le modèle de passage de message, et plus précisément MPI, est largement utilisé dans un grand nombre d'applications scientifiques comme TRIPOLI-4 [JB94], un code de transport monte carlo utilisé dans le contexte de la recherche nucléaire, développé continuellement depuis les années 90. On peut également citer Gromacs [BvdSvD95] (créé en 1995, codé en C), Octopus [CAO+06] (dans les année 2000, codé en Fortran), BigDFT [GNG+08] (depuis 2008, codé en Fortran) qui sont des simulations moléculaires, ou encore Code_Saturne [cod] (depuis 1995, Fortran, 500 000 lignes de code) une simulation d'écoulement et SPECFEM3D [TKL08] (codé en Fortran), un simulateur d'ondes sismiques. Il est difficile d'être exhaustif vu la quantité de simulations existantes basées sur MPI. Il est cependant intéressant de remarquer qu'il existe un grand nombre de logiciels MPI ayant près de 20 ans d'âge mais la vitalité de la norme ne semble pas faiblir car des projets récents, comme Octopus ou BigDFT, ont également choisi d'utiliser MPI.

Le plus souvent, c'est par exemple le cas de BigDFT ou SPECFEM3D, les applications ont été pensées dès leurs conceptions pour être efficaces avec MPI. Cette utilisation massive s'explique par la prédominance des architectures parallèles à base de grappes de calcul, pour lesquelles une approche par passage de message est bien adaptée. De plus, MPI peut être utilisé très efficacement avec le langage Fortran, un langage populaire dans les différentes communautés scientifiques qui développent des simulations. Enfin, les grandes performances de la plupart des implémentions MPI, leurs disponibilités dans la quasi totalité des super calculateurs actuels, et leurs robustesse, rendent MPI quasiment indispensable dans le monde du calcul haute performance.

Un modèle adapté aux architectures homogènes La grande majorité des application bâties sur ce modèle utilise une parallélisation de type SPMD [3] homogène qui consiste à exécuter simultanément sur un ensemble de processeurs un même programme traitant des données différentes – chaque programme traitant une quantité similaire de données. Ainsi, avant l'exécution d'un programme l'ensemble des données du problème est découpé et chacune des parties est attribuée aux processeurs – par l'utilisation, par exemple, d'un mailleur qui va mailler l'espace du problème –.

Cette stratégie est très bien adaptée aux machines parallèles disposant de processeurs homogènes : l'équilibrage de charge se fait naturellement en attribuant aux processeurs une quantité identique de données. Sur des architectures plus hétérogènes, avec des unités de calcul plus ou moins puissantes, comme sur celles comportant des accélérateurs, des problèmes d'équilibrage de charge peuvent survenir.

Une capacité à monter en charge en question Un autre problème qui se pose avec MPI concerne sa capacité à monter en charge. En effet, sur des petites architectures

3. Single Program, Multiple Data

Listing 3.1 – Un programme MPI

```
1  #include "mpi.h"
2
3  int main(int argc, char* argv[])
4  {
5    int    numprocs, myrank ;
6
7    int to_send;
8    /* initialiser to_send */
9
10   MPI_Status status ;
11
12   MPI_Init(&argc,&argv);
13
14   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
15   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
16
17     if (myrank == 0) /* processus 0 envoie a 1 */
18     {
19       MPI_Send(&to_send, 1, MPI_INT, 1, 1, MPI_COMM_WORLD) ;
20     }
21     if (myrank == 1) /* processus 1 recoit de 0 */
22     {
23       MPI_Recv(buff, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
24                &status) ;
25      }
26
27     return 0;
```

parallèles, une communication collective peut se faire assez efficacement. Sur les prochaines architectures, comportant plusieurs centaines de milliers de processeurs, une réduction du volume des communications deviendra cruciale. Des solutions, comme la programmation hybride MPI + threads sont des pistes pour limiter les communications et continuer à utiliser MPI sur les prochaines grandes architectures parallèles.

3.1.2 Mémoire partagée

Les architectures parallèles à mémoire partagée permettent à différents flots d'exécution d'utiliser une même zone mémoire pour communiquer entre eux. Le moyen le plus naturel de programmation de ces architectures est l'utilisation de processus légers (ou *threads*). Ainsi, chaque flot d'instruction est encapsulé dans différents processus légers partageant entre eux le même espace mémoire.

L'utilisation de threads permet une très grande flexibilité mais demande au programmeur de prendre beaucoup de précautions, principalement lors d'accès concurrent à des zones de mémoire partagée. Ainsi, tout un ensemble de primitives de synchronisation (variables de condition, sémaphores) est disponible dans toutes les mises en œuvre de threads.

Listing 3.2 – Un programme OpenMP en Fortran

```
1
2       !\$OMP PARALLEL DO schedule(static) SHARED(accel,veloc,displ)
            PRIVATE(j)
3       do j=1, TAILLE_TAB
4       displ(j = displ(j + deltat*veloc(j + deltatsqover2_bis*accel(j)
5       veloc(j) = veloc(j) + deltatover2_bis*accel(j)
6       accel(j) = 0.
7       end do
```

3.1.2.1 Le standard Posix Thread

Le standard POSIX thread [pos95] définit un ensemble de fonctions et de constantes pour le langage C permettant de contrôler la création, l'exécution et la synchronisation de threads. L'interface de programmation est d'un niveau relativement bas [But97], cela offre donc un contrôle très fin de l'exécution mais engendre quelques difficultés de programmation. Des mises en œuvre de la norme sont disponibles sur un large ensemble de systèmes d'exploitation comme Linux, MacOS, BSD ou Windows.

3.1.2.2 OpenMP

OpenMP est une spécification [omp08] dont la version 1.0 a été mise au point en 1997 par un ensemble d'acteurs impliqués dans le calcul parallèle (IBM, AMD, EPCC...). La spécification définit un ensemble d'annotations à insérer dans un code source C ou Fortran, un ensemble de fonctions et un ensemble de variables d'environnement. L'avantage majeur d'OpenMP par rapport à l'utilisation directe de bibliothèques de threads comme POSIX thread est sa capacité à cacher aux programmeurs les détails de gestion des threads. En effet, pour paralléliser une boucle, une boucle *for* par exemple, il suffit simplement d'indiquer avant celle ci que l'on désire la paralléliser. Le compilateur va alors automatiquement découper les données et créer des threads. Le programmeur a toutefois un certain contrôle sur cette automatisation, il peut choisir comment OpenMP distribue les données dans les threads ou indiquer un nombre de thread à créer. Ce contrôle peut s'effectuer directement dans le code source ou par l'intermédiaire de variables d'environnement définies par les standard. La figure 3.2 montre un exemple de sommes de tableaux en OpenMP adaptés du code de simulation SPECFEM3D [TKL08].

Pour être efficace sur les architectures NUMA actuelles, les threads OpenMP doivent être positionnées avec soin sur les cœurs CPU. Les implantations performantes d'OpenMP, comme celle d'Intel, permettent de positioner précisiment les threads sur les architectures multicœurs (selection d'affinité par socket, par niveau de cache...). Cela demande toutefois d'avoir une bonne connaissance de l'architecture matérielle.

3.1.3 Programmation hybride MPI et threads

Les grappes étant formées de nœuds à mémoire partagée associés en réseaux, il est relativement tentant d'utiliser un modèle de programmation adapté aux architectures

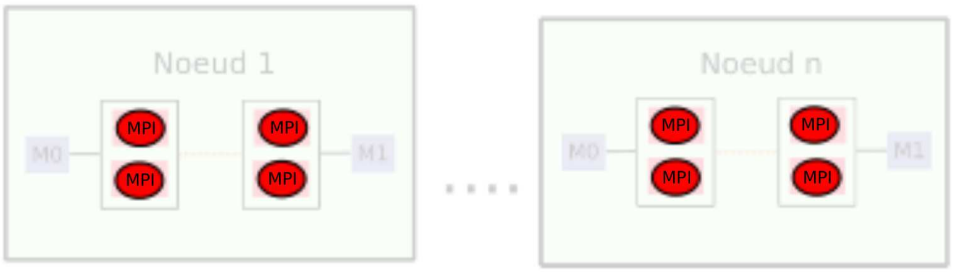


Figure 3.1 – Une tâche MPI par cœur

distribuées associé à un environnement pensé pour les architectures multicœurs à mémoire partagée. Ce type de programmation permet de réduire le surcoût en communication si MPI est utilisé seul, mais il ajoute un coût de création/gestion de threads. Nous considérons ici que la partie thread est assuré par OpenMP.

3.1.3.1 Types de compositions MPI / threads

Il existe trois types d'organisation MPI + threads :

Uniquement MPI Premièrement, on peut citer l'approche qui consiste à utiliser MPI (Figure 3.1) et un thread par tâche. Cela offre l'avantage d'avoir uniquement un paradigme de programmation pour les systèmes distribués et à mémoire partagée. SPECFEM3D ou Gromacs utilisent par exemple cette approche.

Un processus MPI par nœud L'approche suivante consiste en un processus MPI par nœud, chacun associé à un thread OpenMP pour chaque cœur (Figure 3.2). MPI est ici utilisé pour les aspects distribués, et OpenMP gère les aspects liés à la mémoire partagée.

Quelques processus et quelques threads La troisième approche, qui est en quelque sorte une voie médiane entre les deux précédentes se propose d'utiliser quelques processus MPI par nœud contenant des threads OpenMP associés aux cœurs. On peut imaginer par exemple d'utiliser un processus MPI par banc NUMA contenant des threads OpenMP comme sur la figure 3.3. Octopus ou BigDFT, par exemple, peuvent utiliser cette approche.

3.1.3.2 Intérêt de la composition MPI / threads

Divers travaux ont été menés sur cette composition OpenMP / MPI et aucun paradigme ne se démarque véritablement. Dans [CE00], les auteurs ont comparé un modèle pur MPI avec un modèle hybride sur des machines IBM/SP basées sur des processeurs IBM Power 3. Les auteurs ont déterminé que sur les benchmark NAS [BHS+95], l'utilisation de MPI seul était toujours meilleure qu'une conjonction OpenMP / MPI. Des travaux plus récents [WT11] montrent qu'en fonction des benchmarks, c'est l'un ou l'autre des modèles qui prend le dessus.

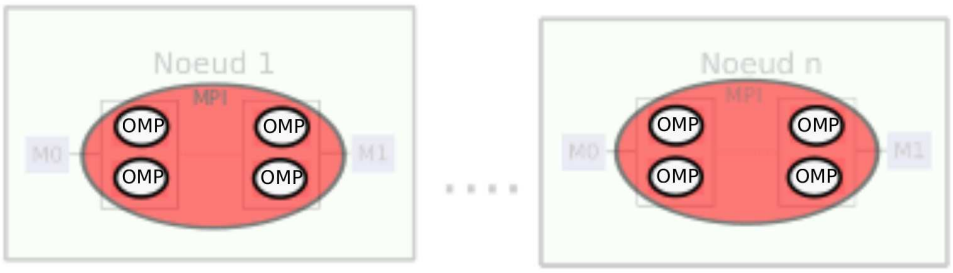


Figure 3.2 – Un processus MPI par nœud et un thread OpenMP par cœur

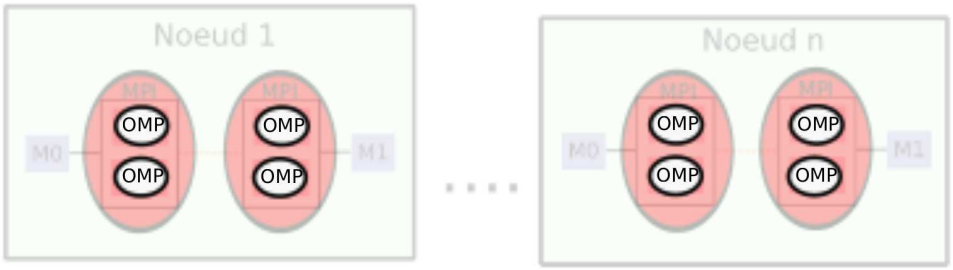


Figure 3.3 – Plusieurs processus MPI par nœud et un thread OpenMP par cœur

La composition MPI + threads est cependant intéressante lorsque l'on considère des machines NUMA. Confiner les threads dans des bancs NUMA distincts évite les transferts distants et peut augmenter les performances de programmes limités par la bande passante mémoire.

3.1.4 Modèle à adressage globale partitionné (PGAS)

Les langages de programmation basés sur le modèle PGAS (Partitioned Global Address Space) exposent un espace mémoire virtuellement unifié à partir d'une architecture distribuée. Il est donc possible de programmer une grappe de calcul à la façon d'une architecture à mémoire partagée. Avec PGAS, est possible de mettre en place des systèmes de communication ou l'on écrit directement dans la zone mémoire d'un autre nœud (communications de type *one sided*). Cette possibilité peut simplifier la programmation ou augmenter les performance par rapport à MPI.

Il existe plusieurs langages mettant en œuvre ce modèle, comme UPC [upc05] (Unififed Parallel C), Chapel [cha11], ou encore Co-array Fortran [NR98]. Ces différents langages définissent des modèles d'exécution quelque peu différents mais utilisent tous le même modèle mémoire.

Ces environnements permettent donc de programmer indifféremment les architectures à mémoire distribuée et à mémoire partagée. Les PGAS sont des modèles de programmation qui sont encore peu utilisés mais semblent prometteurs pour la programmation des futures grappes de calcul.

3.1.5 Synthèse

Les environnements de programmation que nous avons vu jusqu'à présent ont été mis au point à une époque durant laquelle l'on ne parlait pas encore d'accélérateurs et ne possèdent donc pas nativement la capacité de les utiliser [4]. Le support des accélérateurs demande donc, en plus des environnements actuels, d'utiliser de nouveaux environnements, adaptés aux accélérateurs.

Les application actuelles, dont nous avons vu quelques exemples dans cette partie, utilisent globalement toutes MPI et quelques unes commencent à utiliser des stratégies comme OpenMP pour optimiser l'exploitation d'un nœud. Il ne parait pas envisageable de les modifier en profondeur pour les adapter à de nouveau modèles de programmation. La question de l'adaptation des modèles de programmation actuels aux accélérateurs est donc une question cruciale lorsque l'on souhaite permettre aux applications d'utiliser les accélérateurs. Une approche réaliste est donc de conserver les modèles existants qui ont été utilisés pour ces applications et de les adapter, dans la mesure du possible, à ces nouvelles architectures à base d'accélérateurs.

3.2 Programmation des unités spécialisées

Associées à des processeurs généralistes, les unités spécialisées permettent d'obtenir une grande puissance de traitement sur certains types d'opérations. Leurs architectures étant très parallèles et proches des architectures vectorielle, des environnements de programmation spécialisés ont été proposés pour les programmer.

3.2.1 Unité vectorielle intégrée au processeur

La programmation de ce type d'unités vectorielles utilise une extension du jeu d'instruction disponible sur le processeur dans lequel elles se situent. Par rapport au jeu d'instruction original des instructions supplémentaires sont ajoutées pour manipuler des données vectorielles et utiliser de nouveaux registres.

Bien que l'exploitation de ces instructions est importante dans les CPU classiques de type Xeon, s'en passer permet d'atteindre tout de même de bonnes performances. Sur d'autres architectures, comme les Xeon Phi, où chaque cœur possède une importante unité vectorielle, les utiliser est indispensable. Il existe plusieurs moyens de les exploiter :

Vectorisation automatique La majorité des compilateurs récents, comme Intel ICC [BGGT01] ou GCC [Fre], est capable de détecter dans le code source des motifs d'instructions vectorisables. Beaucoup de boucles peuvent être ainsi automatiquement vectorisées par les compilateurs. Ces traitements automatiques peuvent être largement améliorés et de nombreuses recherches sont menées sur ce thème [NRZ06, NH06, BZS10]. Toutes les simulations scientifiques peuvent donc utiliser ces unités vectorielles, plus ou moins efficacement, grâce aux compilateurs. Il est cependant souvent indispensable de retravailler les boucles pour aider le compilateur à les vectoriser. Les compilateurs Intel sont capables,

4. Bien que la prochaine version 4.0 d'OpenMP devrait avoir des mécanismes permettant un support des accélérateurs.

par exemple, d'émettre des rapports de vectorisation permettant d'aider le programmeur à comprendre quel est le niveau de vectorisation du code.

Langages adaptés Des travaux sont menés pour permettre d'exprimer que des constructions sont vectorisables. La norme OpenMP 4 aura par exemple un support pour ce type d'instruction. Intel travaille sur un langage spécialement destiné à la vectorisation : ispc [PM12]. Le langage OpenCL, que nous aborderons plus loin, a également un support des unités vectorielles.

Il est également possible de programmer directement les instructions vectorielles des processeurs grâce à des *fonctions intrinsèques*.

3.2.2 Processeurs accélérateurs

Les processeurs accélérateurs définissent un modèle d'exécution très différent de celui adopté dans les processeurs généralistes. Par conséquent, la programmation de ces processeurs demande des environnements de programmation différents de ceux utilisés classiquement pour programmer les processeurs généralistes.

3.2.2.1 Environnements spécialisés

Les GPU [5] NVIDIA et CUDA En 2007, NVIDIA rend disponible la version 1.0 de l'environnement CUDA [6] [NVI11a]. CUDA définit un modèle de programmation parallèle et des solutions logicielles. Une extension du langage C appelée "C for CUDA" est proposée pour développer des logiciels exploitant le modèle CUDA. La particularité de CUDA est de permettre à des programmeurs familiers avec des langages standards, comme C, d'utiliser la puissance des GPU pour du calcul généraliste. Ainsi, l'utilisation détournée de langages de *shader* (comme gc ou GLSL [LHK+04, OLG+07]) n'est plus nécessaire. L'arrivée de CUDA a eu une importance considérable pour la démocratisation du calcul généraliste sur GPU. CUDA est utilisable pour programmer les GPU NVIDIA à partir de la génération G80.

Le modèle de programmation de CUDA possède une propriété remarquable : il est suffisamment haut niveau pour rendre scalable des programmes sur les différents GPU disponibles. En effet, l'idée de CUDA est de contraindre le programmeur à paralléliser un problème en *blocs*, eux-mêmes parallélisés en *threads*. Ainsi, lorsque le programme crée un nombre suffisant de blocs (plusieurs milliers en général), ceux-ci vont s'exécuter parallèlement sur les différents processeurs disponibles sur le GPU. Pour prendre un exemple, si un programme CUDA met en place 7 000 blocs sur une carte NVIDIA GeForce GT 415M qui comporte un processeur, alors ce processeur traitera successivement ces 7 000 blocs. Si à l'inverse, ce programme est exécuté sur une TESLA C2070 qui comporte quatorze processeurs, chacun d'entre-eux lancera simultanément 500 blocs (7000/14). La figure 3.3 montre un exemple de programme CUDA. La ligne 9 contient la logique permettant de définir le nombre de blocs et la taille de ces blocs, tandis que la ligne 1 contient une fonction exécutable sur le GPU. L'environnement CUDA offre également tout un ensemble de

5. Graphics Processing Unit
6. Compute Unified Device Architecture

Listing 3.3 – Un programme CUDA

```
1  __global__ void kern(int i)
2  {
3    __shared__ float shared_temp[256];
4    int threadId = threadIdx.x;
5
6    shared_temp[threadId] = i;
7  }
8
9  void lanceKernel()
10 {
11   dim3  grille(1000, 1, 1);
12   dim3  bloc(256, 1, 1);
13
14   kern <<< grille , bloc >>> kern(4);
15 }
```

fonctions destinées à mettre en place des transferts mémoire entre les espaces mémoire CPU et GPU.

Les bonnes caractéristiques du modèle CUDA sont intéressantes pour attaquer directement des CPU multicœours au lieu des GPU. Ainsi, des travaux comme MCUDA [SSH08] existent et ont pour but de permettre d'ordonnancer des codes CUDA efficacement sur des CPU. De grandes sociétés se montrent intéressées par ces travaux. PGI, en collaboration avec NVIDIA, développe d'ailleurs un support CUDA pour les architectures processeurs x86 [PGI11].

Les GPU ATI AMD propose également un support de calcul généraliste pour ses processeurs graphiques. La première tentative a été développée avec l'environnement CTM (pour Close To Metal) [Hen07]. Cet environnement repose sur une machine virtuelle DPVM [PSG06] qui propose des abstractions favorisant l'utilisation de processeurs graphiques pour du calcul. AMD a également proposé un environnement basé sur le langage parallèle Brook [BFH+04], qui utilise une notion de flux. La figure 3.4 montre un programme Brook+ définissant trois flux et un kernel appliqués à ces flux.

AMD a abandonné Brook+ et pousse maintenant un environnement de programmation parallèle appelé *AMD Accelerated Parallel Processing* [ATI11] basé sur OpenCL. La norme OpenCL sera évoquée par ailleurs.

Les SPU Cell Pour programmer les processeurs accélérateurs du CELL, IBM fournit tout un écosystème logiciel regroupé dans le Cell/B.E. Software Development Kit [IBM08]. Ce kit contient un ensemble de compilateurs et de bibliothèques logicielles de calcul scientifique (BLAS, LAPACK...). Le niveau de programmation nécessaire pour exploiter le CELL est relativement bas. Il est par exemple nécessaire de gérer manuellement les transferts DMA pour alimenter en données les SPU. Cette spécificité rend la programmation du CELL relativement délicate. Il existe donc beaucoup de travaux, comme [VHHZ08, SFT+09], pour améliorer la programmabilité de ce processeur.

Listing 3.4 – Un programme Brook+

```
1  kernel void sum(float a<>, float b<>, out float c<>)
2  {
3      c = a + b;
4  }
5
6  int main()
7  {
8      int i, j;
9      float a<10, 10>;
10     float b<10, 10>;
11     float c<10, 10>;
12
13     float input_a[10][10];
14     float input_b[10][10];
15     float input_c[10][10];
16
17
18     for(i=0; i<10; i++)
19     {
20         for(j=0; j<10; j++)
21         {
22             input_a[i][j] = (float) i;
23             input_b[i][j] = (float) j;
24         }
25     }
26     streamRead(a, input_a);
27     streamRead(b, input_b);
28
29     sum(a, b, c);
30
31     streamWrite(c, input_c);
32
33 }
```

3.2.2.2 Directives de compilation

Différentes sociétés, comme CAPS Entreprise avec HMPP [DBB07], PGI [Por10] travaillent sur des outils permettant de générer automatiquement du code parallèle adapté aux accélérateurs à partir d'un code séquentiel. Des annotations du code, similaires à celles employées par OpenMP, permettent de définir des régions à paralléliser automatiquement (principalement des boucles)

Ce genre de solutions offre deux avantages principaux :

1. L'indépendance des programmes aux architectures. En effet, lors de la compilation du code source, il est possible de spécifier à HMPP des *architectures cibles*, par exemple CUDA et SSE. Ainsi, lors de la génération du code, HMPP va générer du code exécutable pour toutes les *architectures cibles* définies. Lors de l'exécution, une phase d'exploration a la charge de découvrir les accélérateurs disponibles. S'il y a correspondance entre un accélérateur disponible et une *architecture cible* compilée, le code s'exécutera sur l'accélérateur, sinon il sera lancé sur le processeur central.
2. Intrusivité limitée. La génération automatique du code permet de ne pas apprendre un nouveau langage comme CUDA mais d'ajouter uniquement quelques directives de compilation au code original. Cependant, la gestion des différentes zones mémoire est laissée au programmeur et il est bien souvent nécessaire d'ajouter un code très complexe dans l'application pour gérer les transferts d'une façon efficace.

La problématique de l'utilisation concurrente des différentes ressources CPU et GPU est cependant peu traitée par ces outils. Il faut également noter que ces annotations étant propriétaires un programme annoté devient dépendant d'un compilateur particulier. Cela est toutefois de moins en moins vrai avec l'émergence de standards comme OpenACC[Ope11b]. La figure 3.5 montre un exemple de programme HMPP où le code de la fonction `calc` sera compilé pour la cible CUDA. A la ligne 12, le code sera exécuté sur un accélérateur CUDA s'il est disponible sinon sur le processeur central. La programmation en OpenACC est assez similaire.

D'autres environnements de programmation utilisent des annotations, c'est le cas de StarSS [ABI+09] qui définit un modèle de programmation par tâche et des annotations permettent de définir ces tâches de manière aisée. Cependant, contrairement à HMPP ou OpenACC qui permettent de ne pas modifier la structure du code, StarSS nécessite d'adapter des algorithmes dans un modèle de tâches.

Ces différentes propositions basées sur des directives de compilation et un support du compilateur montrent que de nombreuses tentatives sont effectuées pour fournir aux applications des moyens peu intrusifs pour exploiter les accélérateurs.

3.2.2.3 Le cas OpenCL

L'intérêt des accélérateurs pour le calcul haute performance et la diversité des environnements de programmation nécessaires pour les exploiter a eu pour conséquence le développement d'une norme, appelée OpenCL [OPE11a] pour Open Computing Language. Cette norme spécifie une bibliothèque logicielle et un langage de programmation basé sur le C. OpenCL est une norme suffisamment générale pour permettre à différentes sociétés d'offrir un support pour leurs matériels, accélérateur ou non. Par exemple, NVIDIA et AMD

Listing 3.5 – Un programme HMPP

```
1  #pragma hmpp label1 codelet, args[A].io=out, args [B].io=inout, target=
       CUDA
2  void calc(int n, int A[n], int B[n])
3  {
4   for(int i=0 ; i<n ; i++)
5   {
6    A[i] = B[i] * 2
7    B[i] = B[i] * B[i]
8   }
9  }
10
11 #pragma hmpp calc callsite
12 calc(n, A, B);
```

fournissent un support OpenCL pour leurs accélérateurs graphiques et AMD offre également un support pour ses processeurs généralistes. Ainsi, grâce à la capacité d'OpenCL de compiler des noyaux de calcul directement à l'exécution, un programme pourra être exécuté indifféremment sur ces deux différents processeurs graphiques. Chaque architecture exécutera une version qui aura été automatiquement compilée pour elle. Néanmoins, les architectures matérielles ciblées par OpenCL sont si différentes entre elles, que penser l'exécution efficace d'un même programme sur différentes architectures sans transformation semble illusoire. La pratique montre qu'il est nécessaire d'optimiser un programme vers l'architecture cible. Cependant, seul le noyau de calcul change, le reste peut demeurer identique, ce qui rend ce langage extrêmement prometteur.

Programmation OpenCL Bien que relativement verbeuse, l'utilisation de OpenCL se base sur des concepts clairs. Ainsi, des *command queues*, connectées à un dispositif de calcul (accélérateur ou processeur généraliste) se chargent d'exécuter les commandes qu'elle contient lorsque le dispositif est disponible. Le programme 3.6, représente une configuration où un transfert mémoire et un calcul sont insérés dans une queue afin d'être exécutés. Le code d'initialisation (création de queue...) n'est pas représenté.

3.2.2.4 Support d'autres langages

Les environnements de programmation précédents sont basés principalement sur le langage C. Cependant, d'autres langages possèdent des modules permettant de programmer nativement du code CUDA ou OpenCL. Par exemple, PyCUDA, PyOpenCL [KPL+11] ou CLyther [cly10] ont pour objectif d'autoriser la programmation d'accélérateurs en python. En C++, la bibliothèque Thrust [NVI11b] fournit un nombre important de primitives de programmation parallèle (tris parallèle, réduction ...).

Pour le langage Fortran, il existe peu de moyens natifs pour utiliser CUDA ou OpenCL. Bien souvent, des "liaisons" fastidieuses à mettre en œuvre doivent être développées pour transformer un appel de fonctions Fortran vers une fonction C (contenant du code CUDA ou OpenCL). Il faut aussi prendre en compte dans ce type d'approche que la représentation

Listing 3.6 – Un programme OpenCL

```
1  //mise en queue d'un transfert memoire
2  clEnqueueWriteBuffer(queue,readbuffer,CL_TRUE,0,buffersize,datain,0,
       NULL,NULL);
3
4  //Arguments du kernel
5
6  clSetKernelArg(kernel,0,sizeof(datasize),(void*)&datasize);
7  clSetKernelArg(kernel,1,sizeof(readbuffer),(void*)&readbuffer);
8  clSetKernelArg(kernel,2,sizeof(writebuffer),(void*)&writebuffer);
9
10 //mise en queue du kernel
11 size_t localWorkSize[] = {32};
12 size_t globalWorkSize[]= fshrRoundUp(32,datasize);
13 clEnqueueNDRangeKernel(queue,kernel,1,NULL, globalWorkSize,
       localWorkSize,0,NULL,NULL);
```

mémoire d'un tableau multi dimensionnel est inversé entre un programme Fortran et un programme C.

Une autre stratégie, plus radicale, consiste à convertir entièrement une application initialement en Fortran en C afin d'utiliser CUDA. Les concepteurs de SPECFEM3D ont opté pour cela lors du portage de l'application vers l'utilisation des GPU [KEGM10]. Cette approche n'est envisageable que pour de petites applications, les concepteurs de SPECFEM3D ont porté en C uniquement une partie centrale du code de SPECFEM3D.

Un bon support Fortran s'avère donc nécessaire pour faciliter le portage des grandes applications scientifiques vers l'utilisation des GPU. Des approches comme HMPP peuvent être indiquées car il est possible d'annoter nativement du code Fortran. Pour garder un peu plus de contrôle, PGI propose une autre approche avec CUDA Fortran, qui permet de programmer un programme CUDA en restant exclusivement en Fortran (les kernels cuda peuvent être ainsi écrits directement en Fortran plutôt qu'en C).

3.2.3 Synthèse

Utiliser les accélérateurs dans un code pose un certain nombre de questions. La première consiste à s'interroger sur les opérations qui sont susceptibles d'être accélérées. Il est difficile de convertir la totalité d'une grande application sur un accélérateur, une démarche raisonnable consiste plutôt à porter uniquement quelques parties de l'application. Pour cela, une étude approfondie du code doit être entrepris et les sections intensives en calcul isolées. Des sociétés comme CAPS avec HMPP fournissent des outils d'aide à l'analyse de programme permettant de déterminer simplement quelles parties du code peuvent être accélérées. D'autres outils, comme Intel Trace Analyser and Collector, que nous verrons plus loin, permettent d'avoir une vue globale sur une application parallèle pour y chercher, entre autre, les parties à accélérer.

Une autre question est reliée à la programmation de ce type de matériel. Les accélérateurs exposant au programmeur une grande complexité, il est important d'avoir une

connaissance fine de ces architectures, au niveau des accès mémoire ou au niveau du contrôle de l'exécution pour atteindre de bonnes performances. Il faut mettre en place des algorithmes qui exploitent le mieux possible les caractéristiques techniques des accélérateurs pour en tirer de bonnes performances. Le choix de l'environnement de programmation est ici prépondérant : des environnements à base de directives, comme HMPP ou OpenACC, permettent de laisser la génération de code accéléré au compilateur mais empêchent l'utilisation d'optimisations poussées qui sont accessibles via un environnement spécialement conçu pour un accélérateur comme CUDA.

Enfin, l'intégration d'un code accéléré au sein d'une application doit également être considérée. Le code programmé pour un accélérateur doit être intégré dans l'application. Encore une fois, des environnements à base de directives peuvent simplifier cet aspect, mais le manque actuel de mise en œuvre de spécifications standards bien établies rend difficile de bâtir un projet sur un compilateur particulier. Cet aspect sera précisé dans la partie suivante mais il est déjà clair qu'un support Fortran robuste est indispensable pour rendre les accélérateurs utilisables dans les applications scientifiques actuelles.

3.3 Programmation hybride avec accélérateur

Les environnements présentés jusqu'à présent permettent soit d'utiliser des grappes homogènes (c'est le cas de MPI par exemple) soit d'utiliser les accélérateurs (c'est le cas de CUDA ou HMPP). Utiliser une grappe hybride demande d'utiliser ces deux types d'environnement simultanément. C'est l'objet de cette partie.

3.3.1 MPI, Threads et accélérateurs

MPI étant l'environnement le plus utilisé lorsque l'on parle d'application scientifique, nous nous concentrons ici sur les moyens employés par une application MPI lorsqu'elle utilise les GPU.

3.3.1.1 Exploitation des accélérateurs avec MPI

Dans une architecture homogène comportant uniquement des CPU de puissance identique, les données d'un programme SPMD sont partitionnées de telle sorte que chaque processeur traite une quantité de données qui permette à tous les processeurs de finir leurs exécutions simultanément. L'équilibrage de charge dans notre cas est statique et résolu lors du partitionnement du problème avant l'exécution du programme.

L'ajout d'accélérateurs complexifie ces considérations, nous rajoutons en effet un type d'unité de calcul et les stratégies employées pour le cas d'une architecture homogène doivent être revues. Comment en effet doit-on partitionner le problème ? Faut-il, par exemple, considérer qu'il existe des tâches MPI utilisant des groupes d'exécution composés de couples {CPU + accélérateur} et d'autres tâches utilisant uniquement des CPU ? Dans ce cas, le partitionnement du problème devra prévoir d'allouer plus de données aux tâches MPI utilisant les processeurs et les accélérateurs et moins aux tâches utilisant des processeurs seuls. Ce calcul de partitionnement devra être fait pour chaque machine car en fonction des modèles de processeur, la puissance relative entre un CPU et un GPU

est variable. Une autre approche peut consister à considérer uniquement des tâches MPI utilisant les couples {CPU + accélérateur}. Cette approche a l'avantage de pouvoir utiliser le partitionnement destiné aux architectures homogènes (chaque couple {CPU + accélérateur} a une puissance identique). Cependant, sur la majorité des architectures hybrides où le nombre d'accélérateurs est inférieur à celui des CPU, cette approche manque de souplesse et le nombre de tâches MPI utilisées sera dépendant du nombre d'accélérateurs. Si une tâche MPI ne peut utiliser qu'un CPU, elle sous-utilisera l'exploitation de la machine.

CPU et accélérateurs pouvant être utilisés simultanément, il faut choisir quel type de calcul chacune des deux ressources aura la charge de traiter.

3.3.1.2 Répartition des calculs entre CPU et accélérateurs

Certains calculs sur les CPU et d'autres sur les accélérateurs Une première approche consiste à utiliser les CPU et les accélérateurs sur des calculs différents. Les CPU s'occupent, par exemple, des communications MPI et de certains calculs qui n'ont pas été portés sur accélérateurs et les accélérateurs traitent le reste des opérations. L'approche utilise des ressources CPU et accélérateur, mais a le désavantage d'exploiter d'une manière assez limitée la possibilité d'utiliser simultanément CPU et accélérateurs. Toutefois, c'est une approche efficace lorsque un programme n'a pas de « points chauds », c'est-à-dire un endroit qui concentre la majorité du temps d'exécution. Il est ainsi possible de porter progressivement une application sur accélérateur, en rajoutant au fil des évolutions du portage des calculs accélérés. Il peut aussi s'avérer qu'un calcul initialement sur CPU soit totalement inefficace sur un accélérateur : il vaut donc mieux le laisser sur le CPU. Un programme est donc ici une succession de parties accélérées et non accélérées.

Les versions accélérées de Gromacs et Octopus utilisent cette approche. Cependant, Gromacs n'est pas capable de paralléliser son exécution lorsque le code GPU est activé et Octopus ne peut pas utiliser de système multi-GPU.

Des calculs simultanés sur CPU et accélérateur Au lieu d'utiliser CPU et accélérateur sur différents calculs, il est peut être intéressant d'exploiter les deux ressources simultanément pour utiliser entièrement la puissance de calcul fourni par le matériel.

Il est possible de réaliser cela en utilisant uniquement MPI et un environnement pour accélérateur, mais la difficulté d'effectuer un équilibrage de charge entre les différentes ressources de calcul a donné naissance à un grand nombre de supports exécutifs basés sur des modèles de programmation qui peuvent être éloignés de ceux utilisés par les applications SPMD. Quelques supports exécutifs sont décrits dans la partie suivante.

3.3.2 Équilibrage de charge CPU / accélérateur

Les différents processeurs (cœurs CPU et accélérateurs) d'un nœud hybride pouvant travailler simultanément, l'équilibrage de charge entre ces deux unités d'exécution est un problème très étudié. Différentes stratégies permettant de réaliser cette opération sont ici présentées.

3.3.2.1 Équilibrage de charge statique

En utilisant les environnements logiciels présentés précédemment, il est tout à fait possible d'utiliser toutes les ressources d'une grappe hybride. Par exemple, la mise en œuvre du benchmark LINPACK [Fat09], NVIDIA utilise une méthode qui permet d'utiliser tous les CPU et tous les GPU des nœuds d'une grappe hybride. Pour ce faire, la matrice utilisée par LINPACK est découpée en plusieurs parties. Dans le cas où il y aurait un GPU et plusieurs CPU, la matrice serait découpée en deux parties de taille inégale. Une des parties s'exécutera sur le GPU en faisant des appels à la bibliothèque NVIDIA CUBLAS [NVI11a], laquelle met en œuvre des calculs d'algèbre linéaire sur GPU. Le second bloc de la matrice sera quant à lui calculé sur les CPU grâce à la MKL [Int11a], une bibliothèque de calcul scientifique multithreadée avec OpenMP. Il suffit donc de spécifier à la MKL le nombre de cœurs CPU disponibles pour pouvoir tous les utiliser. Lorsque le GPU et les CPU sont venus à bout de leurs calculs, les deux morceaux de la matrice sont combinés afin d'obtenir le résultat final.

Tout le problème de cette approche est la découpe de la matrice, il faut placer de manière statique une certaine quantité de données sur les GPU et sur les CPU car leurs performances en calcul sont différentes. De plus, un changement de matériel (par exemple des cœurs CPU plus puissants), impose de recalculer manuellement une nouvelle répartition.

3.3.2.2 Équilibrage de charge dynamique

Il existe beaucoup de propositions destinées à utiliser de manière cohérente les processeurs généralistes et les accélérateurs. Nous en présentons quelques-unes dans cette partie.

Charm++ L'environnement parallèle Charm++ [Kal90], a pour but de proposer des mécanismes évolués d'équilibrage de charge pour les programmes parallèles. Des applications scientifiques basées sur Charm++ ont été développées, comme NAMD [PBW+05]. Charm++ a été adapté à l'usage des accélérateurs [Wes08] (GPU et Cell).

Une restructuration complète des applications est nécessaire pour les convertir au modèle Charm++, ce qui rend cet environnement plutôt envisageable pour de nouvelles applications. La couche logicielle AMPI peut être une solution pour exploiter les possibilités de Charm++ directement depuis un programme MPI mais il n'existe pas à notre connaissance d'applications hybrides utilisant AMPI pour exploiter des accélérateurs.

StarPU L'objectif de StarPU [ATNW11, Aug09] est d'ordonnancer efficacement des tâches sur des CPU multicœurs et sur des accélérateurs. Pour cela, des tâches contenant du code CPU et des tâches contenant du code accélérateur (CELL, GPU) doivent exister : elles sont organisées selon un graphe de dépendance qui permet à StarPU de déterminer quelles tâches peuvent être exécutées à un instant donné. Tout un ensemble d'ordonnanceurs est fourni et une API permet à l'utilisateur de développer lui même une nouvelle politique d'ordonnancement.

StarPU définit une mémoire partagée virtuelle (ou DSM pour Distributed Shared Memory) qui permet d'effectuer les transferts mémoire centrale $\leftrightarrow$ accélérateur de manière

automatique.

Harmony Harmony [DY08, DY10] propose une solution d'équilibrage de charge basée sur une technique d'exécution similaire à celle trouvée dans les processeurs superscalaires. Un calcul est implémenté sous forme de *kernel* qui peut être exécutable sur différents types de processeur. Harmony génère en fait le code CPU à partir du code intermédiaire PTX que produit le compilateur CUDA. En fonction des dépendances de données, Harmony va exécuter ces kernels sur les processeurs disponibles sur la machine. Une exécution spéculative est aussi proposée, laquelle permet d'optimiser encore les performances.

Qilin Qilin [LHK09] est basé sur une approche qui consiste à découper un calcul en deux parties. Pour utiliser Qilin, un problème doit être exprimé avec une partie CPU (Intel TBB) et une partie GPU (CUDA). Une première exécution d'un programme utilisant Qilin permet de calibrer un système d'estimation du temps d'exécution destiné à déterminer la puissance de calcul des CPU et des GPU. Avec cette information, la bonne quantité de calcul à assigner aux différents processeurs peut être déterminée par Qilin pour la prochaine exécution.

Les supports exécutifs définis ci-dessus offrent de bonnes performances et montrent qu'un équilibrage automatique des calculs entres les CPU et les accélérateurs est important pour exploiter les architectures hybrides. Cependant, les modèles de programmation qu'ils définissent n'ont pas été prévus pour interagir avec des modèles de programmation comme MPI et OpenMP. Ils peuvent etre également relativement difficiles à mettre en place dans des applications existantes utilisant MPI et des threads : il faut en effet créer des objets communicants dans le cas Charm++ ou restructurer le code pour utiliser des tâches avec StarPU et StarSS.

Par conséquent, les concepteurs de l'application DL_POLY [KTS10], ont préféré mettre au point une technologie totalement liée à leur application pour réaliser des calculs distribués sur les CPU et les accélérateurs plutôt que d'utiliser un support exécutif spécialisé. Mise à part Charm++ qui cible les grandes applications, ces supports exécutifs sont plutôt destinés à la conception de bibliothèques logicielles réutilisables : StarPU a été expérimenté par exemple sur des calculs d'algèbre linaire [AAD+10, AAD+11a, AAD+11b].

3.3.3 Synthèse

L'intégration des accélérateurs dans les applications nécessite l'utilisation d'environnements logiciels spécifiques (comme CUDA ou OpenCL) à ajouter aux environnements plus classiques de type MPI ou OpenMP. L'hétérogénéité de puissance induite par les grappes hybrides – il existe des processeurs plus ou moins puissants – demande des réflexions poussées aux concepteurs d'applications pour adapter celles-ci aux grappes de calcul hybrides.

Conserver un modèle de programmation MPI adapté à l'utilisation des accélérateurs est donc crucial et le manque dans ce domaine impose aux grandes applications MPI de concevoir leurs propres solutions ou de limiter leurs exploitations du matériel. Pour des bibliothèques de calcul, en revanche, comme certaines spécialisées dans des calculs

matriciels, les contraintes sont beaucoup moins fortes et les repenser entièrement pour leur permettre d'utiliser un support exécutif performant est tout à fait envisageable. La bibliothèque d'algèbre linéaire MAGMA a par exemple été adaptée [AAD+10] pour être utilisée avec StarPU.

Il est donc tout à fait possible de produire, pour chaque partie du code à accélérer, des bibliothèques utilisant un support exécutif. Cependant, d'autres difficultés sont à prendre en compte et reviennent encore à l'intégration de ces supports exécutifs dans un environnement MPI où plusieurs processus peuvent cohabiter dans un même nœud. Ce problème est peu traité, mis à part pour StarPU qui possède des fonctions permettant de faire cohabiter plusieurs instances de StarPU mais qui oblige le programmeur à choisir une seule tâche MPI par nœud qui utilisera StarPU.

3.4 Observation de programmes hybrides

La multiplication des environnements et des modèles de programmation, l'utilisation de bibliothèques spécialisées réalisant des opérations automatiques, ou l'asynchronisme d'exécution des processeurs de grappes hybrides rend la compréhension d'une exécution hybride très difficile à réaliser *a-priori*.

Il est donc important d'avoir des outils appropriés pour comprendre l'exécution d'une application afin de l'améliorer.

3.4.1 Qu'observer ?

Les informations observables peuvent être séparées entre celles fournies par le matériel (compteurs matériels des processeurs), et celles fournies par les couches logicielles (MPI, OpenMP).

La plupart des fabriquants de processeur [Int11e, AMD09, NVI11a], accélérateur ou non, fournissent un support permettant d'accéder à des compteurs matériels. Ces compteurs permettent d'obtenir des informations très fines pour comprendre l'exécution d'un programme. Sur les processeurs généralistes, il est par exemple possible de déterminer le nombre d'erreurs de prédiction de branchement ou le nombre de défauts de cache. Sur les processeurs accélérateur de type NVIDIA, des compteurs permettent, entre autre, de déterminer si les accès mémoire sont efficaces (coalescés ou non), ou si les différents WARP s'exécutent sans divergence. Grâce à ces nombreux compteurs (il en existe plusieurs centaines sur les processeurs Intel Nehalem), il est possible de comprendre de manière très précise ce qui se passe durant une exécution.

À un niveau plus haut, celui des couches logicielles, avoir des informations sur le comportement d'un programme est capital. La multiplication des environnements de programmation rend en effet très difficile la compréhension de problèmes si des indices précis ne sont pas disponibles. Pour les programmes MPI, il est essentiel d'observer si l'équilibrage de charge est bon entre les différentes tâches MPI (pour éviter de laisser des nœuds inutilisés tandis que d'autres seront surchargés). Il est très intéressant également d'analyser les temps de synchronisation entre les tâches et les différents recouvrements possibles entre un calcul et une communication. Pour les threads, outre l'équilibrage de charge et

les temps de synchronisation, les coûts de création de threads sont un facteur important à prendre en considération.

Pour les accélérateurs, la difficulté provient de la possibilité de les utiliser de manière asynchrone. Saisir si une communication a bien recouvert un calcul, si un calcul CPU a pu s'exécuter simultanément avec un calcul accéléré est ici d'une première importance.

3.4.2 Comment observer ?

L'observation du comportement d'un programme commence par l'enregistrement d'information, ou trace d'exécution. Les traces peuvent être générées tout au long de la vie d'un programme dans le but d'avoir une vision temporelle de son activité. Il est aussi possible d'avoir une information synthétique à la fin d'un programme.

Intrusivité La création de traces d'exécution peut être plus ou moins intrusive dans le programme. Il existe trois types d'intrusivité, orthogonales entre elles. Tout d'abord l'intrusivité "code source", qui consiste à évaluer les changements à opérer dans le code source même du programme pour effectuer un traçage. Ensuite l'impact du traçage sur les performances, ou intrusivité "performance". Enfin, l'intrusivité "système" lorsque des bibliothèques logicielles ou des pilotes de périphérique doivent être ajoutés. L'idéal serait bien entendu de n'avoir rien à ajouter dans le code source avec un impact nul sur les performances.

Format de trace La succession des évènements nécessaires au traçage d'application massivement parallèle doit être enregistrée dans une base de données. Dans ce but, plusieurs tentatives ont été réalisées pour mettre au point des formats de trace. Il existe un grand nombre de ces formats adaptés aux applications parallèles, comme TAU trace format [SM06], Structured Trace Format (STF) [INT11c], le format de trace Pajé [dKSB00], ou le format OTF [MN06]. Ces formats sont tous basés sur une notion d'évènements associés à un temps. Les évènements peuvent avoir différentes sémantiques, et peuvent représenter des information très variées comme la fin d'une communication point à point ou un appel de fonction.

Généralement, ces traces sont enregistrées directement sur les disques. Pour éviter les problèmes d'accès concurrents et de congestion, une multitude de fichiers est créée pour être ensuite fusionnée entre eux à la fin de l'exécution. Les formats OTF et STF ont pris en charge cet aspect directement lors de leur conception.

Trace des compteurs matériels La récupération d'informations des compteurs matériels peut se faire grâce à des outils intégrés, comme Intel Vtune [INT11d], ou avec l'utilisation de bibliothèques logicielles comme PAPI [pap]. Le coût engendré par la lecture des compteurs matériels est faible, de l'ordre de quelques pourcents sur le temps total d'exécution d'un programme.

Trace applicative Plusieurs techniques sont disponibles pour tracer directement une application. La moins intrusive, du point de vue du code source, est la surcharge des fonctions. Une bibliothèque peut intercepter un appel de fonction – par exemple MPI –, enre-

gistrer un évènement dans une trace et appeler la véritable fonction. C'est une technique très commune, la bibliothèque MPI définit d'ailleurs une interface, appelée PMPI permettant de faciliter ce traçage. La surcharge est utilisée par Intel Trace Collector [INT11c], VampirTrace [MKJ+07] ou EZtrace [TRF+11] qui sont des environnements logiciels permettant de capturer MPI, OpenMP, et même CUDA/OpenCL pour VampirTrace. Ces environnements enregistrent leurs traces respectivement en STF et OTF.

L'autre technique de traçage est l'ajout d'informations dans le code source de l'application. Bien qu'intrusive, cette approche permet d'observer des informations impossibles à capturer avec des outils génériques, comme le temps total d'un calcul comportant plusieurs fonctions. Intel Trace Collector [INT11c] ou VampirTrace [MKJ+07] supportent ce type de fonctionnalité.

3.4.3 Interprétation des résultats et analyse des applications

L'interprétation des résultats de très bas niveau produits par les compteurs matériels et ceux produits par les couches logicielles peut être compliquée. Il est en effet difficile de déterminer pourquoi une application pose des problèmes de performance (problème de cache ? Problème de bande passante mémoire ?). Ainsi, des méthodologies [Dro08] existent et proposent d'aider à savoir quels compteurs analyser parmi l'énorme quantité d'information disponible.

Pour rendre cette tâche plus abordable, Intel a rendu disponible VTune Amplifier [INT11d]. A la première exécution d'un programme avec VTune, le logiciel sélectionne un ensemble large de compteurs matériels généraux. Après cette exécution, un rapport est produit et permet d'orienter l'utilisateur vers une sélection des compteurs à analyser. Bien que cette approche simplifie grandement l'analyse de performance sur processeur Intel, une connaissance approfondie de l'architecture des processeurs est toujours nécessaire. Cet outil permet également d'étudier une exécution OpenMP en visualisant les temps de synchronisation et de création de threads.

Après un traçage de l'application (avec VampirTrace, Intel Trace Collector, ou EZTrace), des outils comme *Intel Trace Analyser And Collector (ITAC)* [INT11b], *Vampir* [vam] ou *Visual Trace Explorer (ViTE)* [INR10] offrent à l'utilisateur la possibilité d'avoir une vue globale sur une application. La Figure 3.4 montre un exemple de l'observation de l'application BigDFT avec ITAC sur laquelle le code source a été instrumenté pour observer des sections particulières du programme (`Precondition, ApplyLocPotKin`). Il est ainsi possible de voir globalement différentes phases des huit tâches MPI lancées pour l'application.

3.4.4 Construire ses propres outils d'observation

Pour avoir un maximum de flexibilité et se passer d'outils lourds comme Intel Trace Collector, il est possible d'écrire directement des évènements dans un format de trace particulier. Ainsi, OTF propose une bibliothèque logicielle légère qui met en œuvre une API permettant de générer des traces. Des outils sont fournis et permettent de manipuler les fichiers, par exemple, un utilitaire appelé `otfmerge` peut être utilisé pour fusionner les traces des différentes contributions. Il existe également des interfaces de plus haut niveau

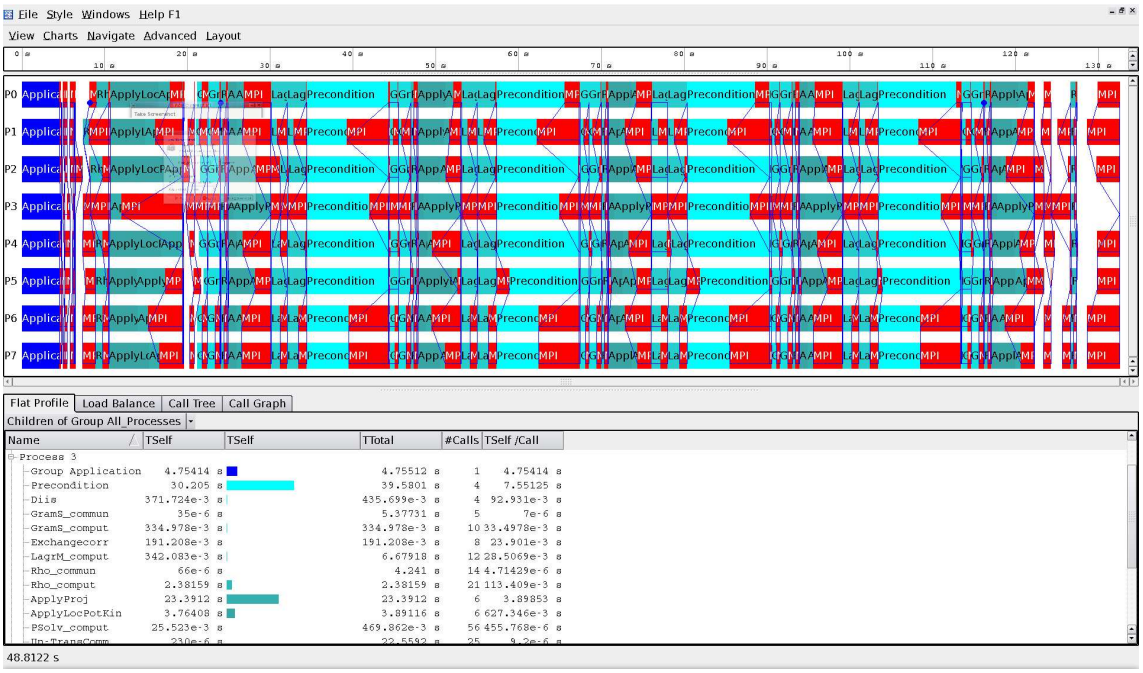


Figure 3.4 – Capture d'écran d'une exécution d'Intel Trace Analyser

(GTG, Generic Trace Generator [gtg]) permettant d'avoir une API unifiée et générée indifféremment du Pajé ou de l'OTF.

Pour visualiser les traces ainsi générées, il suffit d'utiliser un outil capable de les interpréter comme Vampir ou Vite.

3.5 Conclusion

L'utilisation efficace des grandes grappes de calcul hybrides demande d'utiliser différents modèles de programmation qui ont bien souvent été conçus sans prendre en compte une utilisation conjointe.

Des recherches ont été menées depuis quelques années pour tenter d'exploiter l'ensemble des ressources disponibles en utilisant donc dans un même programme différents environnements logiciels (par exemple CUDA associé MPI ou OpenMP). Quelques grandes applications scientifiques ont été portées sur des accélérateurs et des supports exécutifs ont été conçus dans le but d'utiliser efficacement CPU et accélérateur. Les grandes applications scientifiques qui utilisent les accélérateurs sont toutefois assez rares et certaines d'entre elles, comme Gromacs, ont un support très limité des accélérateurs.

Un nombre important de supports exécutifs et de modèles de programmation demande de profonds changements dans les applications pour être utilisables. Ces changements sont généralement inenvisageables car beaucoup d'applications sont développées depuis plusieurs décennies et ont plusieurs centaines de milliers de lignes de code. Ces supports exécutifs étant plutôt bien adaptés pour la conception de bibliothèques logicielles, il peut être tentant de modifier partiellement les grandes application pour qu'elle utilisent ces

bibliothèques hybrides. Cependant, il est difficile d'imaginer que toutes les applications scientifiques puissent se résumer à appeler des fonctions de bibliothèques standards, il semble pertinent de penser que des algorithmes exprimés dans des applications doivent être aisément hybridés "à la main" et intégrables dans un environnement MPI.

Nous avons travaillé à la proposition de modèles de programmation compatibles avec MPI et utilisables dans les applications scientifiques actuelles. Ainsi, ces modèles ont été conçus pour être peu intrusifs dans les applications actuelles. Ainsi, plutôt que d'utiliser des modèles basés sur des tâches, comme StarSS ou StarPU, qui peuvent être difficiles à intégrer dans un environnement MPI, nous avons voulu proposer des modèles de programmation basés sur des concepts bien établis dans le monde du calcul haute performance, comme les threads, afin de faciliter la programmation hybride CPU et accélérateur.

Ainsi, les contributions de cette thèse, décrites dans les chapitres suivants, sont positionnées autour de quatre axes. Le premier (i), consiste à proposer des moyens pour conserver une parallélisation SPMD homogène avec MPI tout en utilisant des accélérateurs. Le deuxième (ii) consiste à construire un modèle de programmation, basé sur des threads, gérant CPU et accélérateurs. Le troisième (iii) axe est consacré à l'observation de programmes hybrides. Enfin, le quatrième axe consiste à évaluer ces propositions avec l'aide d'applications scientifiques.

Deuxième partie

Contributions

Chapitre Quatre

Modèles de programmation et d'exploitation hybrides

Sommaire

Dans les chapitres précédents, nous avons vu que l'ajout d'accélérateurs dans les grappes de calcul nécessite d'adapter les applications afin qu'elles puissent utiliser ces nouvelles architectures. L'ajout d'accélérateurs complexifie la programmation et il n'y a pas aujourd'hui de solutions clairement établies relatives à l'utilisation de ces accélérateurs dans les simulations scientifiques.

Les contributions de cette thèse – décrites dans ce chapitre sous l'angle des modèles de programmation – consistent à proposer des abstractions utilisables dans les modèles de programmation actuels, permettant aux applications scientifiques d'utiliser ces architectures hybrides.

4.1 Objectifs

Nous proposons, dans cette thèse, d'étendre les modèles de programmation classiques utilisés par les applications scientifiques vers des modèles hybrides. La démarche suivie peut être qualifiée de « descendante » : nous sommes partis d'applications aujourd'hui disponibles et avons cherché à comprendre, à partir de celles-ci, comment utiliser au mieux le matériel hybride. Nous nous concentrons donc ici sur l'utilisation des accélérateurs pour des codes de simulations déjà existants.

Nous détaillons les objectifs des modèles que nous proposons. Trois objectifs principaux ont guidé nos choix et sont présentés dans cette section. Le premier est consacré aux moyens permettant de donner aux applications scientifiques existantes la possibilité d'utiliser les machines hybrides. Le second s'intéresse à l'exécution du code hybride sur les ressources matérielles des machines. Le dernier objectif concerne la visualisation des exécutions hybrides.

4.1.1 Adapter les applications scientifiques pour les grappes hybrides

Nous avons vu que beaucoup d'applications scientifiques utilisent des modèles de programmation bien établis comme le passage de messages avec MPI ou la programmation par threads, voire les deux dans le cadre de la programmation hybride. De plus, nous avons également vu que de nouvelles simulations en phase de développement utilisent également MPI : l'environnement MPI est aujourd'hui incontournable lorsque l'on parle de simulations scientifiques.

L'objectif de ce chapitre est de proposer des modèles de programmation permettant aux applications scientifiques existantes, basées sur MPI, d'utiliser les architectures hybrides. Ces modèles sont bâtis sur des abstractions présentées dans la suite de ce chapitre.

4.1.1.1 Abstractions des architectures matérielles hybrides

Nous définissons dans cette thèse des abstractions destinées aux architectures hybrides. Elles ont trois buts :

Le premier consiste à permettre aux applications de réutiliser leurs stratégies de parallélisation mises au point lors de leur développement. Vu que nous nous intéressons à des applications existantes, la majeure partie d'entre elles a été développée pour des architectures homogènes. Pour cela, les abstractions que nous proposons permettent de rendre disponible au programmeur une architecture plus homogène que l'architecture sur laquelle le code s'exécute.

Les architectures hybrides sont multiples, par exemple, le nombre de cœurs et d'accélérateurs peut fortement varier d'une architecture à l'autre. Pour éviter aux applications d'être modifiées lors d'un passage d'une architecture à une autre, le deuxième but de nos abstractions et de donner aux applications la possibilité d'être utilisables sur différentes machines, nous voulons pour ainsi dire les rendre plus portables.

Un défi, encore aujourd'hui, est d'utiliser de manière efficace toutes les ressources d'une machine hybride (CPU et accélérateur). Différentes stratégies ont été discutées dans l'état de l'art et nous proposons ici, comme troisième but de nos abstractions, de permettre aux simulations scientifiques d'exploiter toutes les ressources de calcul d'une grappe.

4.1.1.2 Fabriquer des modèles peu intrusifs

Du point de vue des modèles de programmation, nous définissions la non-intrusivité comme la propriété de certains modèles à être intégrables facilement dans un modèle de programmation existant. Dans notre cas, les modèles existants sont ceux utilisés par les simulations scientifiques actuelles. Les applications utilisant en majorité une approche

SPMD [1] avec MPI, nous proposons des modèles utilisables avec cette stratégie de parallélisation. Pour adapter de tels modèles, plutôt que d'en créer de nouveaux de toutes pièces, nous proposons des extensions aux modèles utilisés par les applications scientifiques. Ces extensions se basent sur nos abstractions.

Le code des applications scientifiques est prévu pour avoir une durée de vie beaucoup plus importante que les architectures sur lesquelles elles sont exécutées. Nous avons déjà établi que, selon nous, la modification en profondeur de ce type d'applications doit être impérativement limitée. Ainsi, notre but ici est de permettre à ces applications d'utiliser nos modèles peu intrusifs pour qu'elles puissent exploiter les architectures hybrides.

4.1.2 Déploiement des calculs sur le matériel

Les programmes définis à l'aide d'abstractions sont amenés à être exécutés sur le matériel hybride. Il est donc nécessaire d'utiliser le mieux possible les ressources matérielles pour exécuter ces programmes. Le déploiement des calculs consiste donc à les exécuter sur les différentes ressources matérielles lors des différentes phases d'une simulation scientifique.

Des calculs définis avec des abstractions ne sont pas directement déployables sur le matériel. Ainsi, pour déployer ces calculs, il faut passer par une étape intermédiaire dans laquelle le lien entre les abstractions et le matériel est effectué. Pour mettre en œuvre ce déploiement nous proposons des supports exécutifs. Ce sont ces supports exécutifs qui ont la charge de faire les choix de déploiement et pourront ainsi utiliser des caractéristiques avancées du matériel hybride pour optimiser les performances.

Dans le cas des accélérateurs de type GPU, les caractéristiques sur lesquelles nous avons travaillé sont le recouvrement des calculs par des transferts mémoire et l'utilisation conjointe CPU – accélérateur. En recouvrant les calculs et les transferts, nous cherchons à limiter le poids des transferts mémoire. L'exécution simultanée CPU - accélérateur permet d'exploiter toutes les ressources de calcul de la machine en évitant d'en laisser certaines inactives.

4.1.3 Observer et comprendre les exécutions hybrides

Comprendre le déroulement d'une exécution est essentiel lorsque l'on parle de calcul haute performance. Cela permet, en effet, de détecter des problèmes logiciels lorsque un calcul ne se déroule pas comme prévu, et cela permet également d'optimiser les performances s'il est possible d'établir qu'un point particulier d'un programme peut être accéléré. Alors que trouver des erreurs logiciels se fait généralement en-ligne, c'est-à-dire que l'on suit pas à pas le fil d'exécution pour trouver le point exact de l'erreur, l'optimisation de performance se fait plutôt après l'exécution d'un programme. Ainsi, un programme doit être instrumenté ; il s'agit en quelque sorte d'installer des sondes dans le programme pour que celui-ci puisse enregistrer son activité dans des traces d'exécution. Un autre programme a la charge d'interpréter ces traces pour, par exemple, les afficher de manière graphique et montrer à l'utilisateur le déroulement de l'exécution de son programme pour qu'il puisse

1. Single Program, Multiple Data

isoler les parties qui peuvent être optimisées ou corrigées. C'est ce type d'approche que nous avons évalué.

Les accélérateurs et les CPU ont chacun des environnements complets permettant d'enregistrer des mesures de performance – par exemple des compteurs matériels –. Notre but est plutôt de mettre en évidence les interactions entre les différentes ressources de calcul et les différentes zones mémoire d'une machine hybride. Ainsi il s'agira d'isoler les différents recouvrements de transfert et les éventuels problèmes d'équilibrage de charges entre CPU et accélérateurs. Ces différentes informations pourront ensuite être réutilisées par le programmeur pour optimiser son application.

4.2 Virtualisation des ressources hybrides

Nous définissons, à présent, de manière plus précise les abstractions et les modèles de programmation résultant que nous avons proposés dans cette thèse. Nous décrivons deux abstractions et deux modèles.

4.2.1 Abstractions proposées

Les deux abstractions proposées sont basées sur une notion de virtualisation des ressources de calcul. La première propose d'utiliser des accélérateurs virtuels. L'hétérogénéité de puissance provenant du nombre différent de cœurs CPU et d'accélérateurs, l'utilisation d'accélérateurs virtuels permet d'utiliser un nombre quelconque d'accélérateurs. Une application peut ainsi utiliser un environnement plus homogène tout en étant moins attachée à une configuration matérielle spécifique. La seconde abstraction est basée sur l'ajout des ressources CPU à la virtualisation des accélérateurs. Ainsi, toutes les ressources de calcul de la machine (CPU et accélérateurs) sont virtualisées. Pour ce faire, une unité d'exécution abstraite se composant du couple {CPU + accélérateurs} est définie. En se basant sur cette abstraction, nous développons un modèle de programmation permettant de mettre en place des calculs hybrides : dans notre cas, un calcul hybride consiste à utiliser les accélérateurs et les processeurs généralistes sur le même calcul, en utilisant nos objets virtuels {CPU + accélérateurs}. Le programmeur pourra ainsi créer des objets utilisant ces deux ressources de calcul simultanément.

4.2.2 Virtualisation des accélérateurs

Pour exploiter efficacement les architectures multicœurs hybrides, telles que nous les avons présentées au chapitre 2, nous avons choisi d'évaluer une stratégie où les accélérateurs sont virtualisés dans un nœud hybride. Il ne s'agit pas ici d'adapter des machines virtuelles comme VirtualBox [2] ou VMware [3] aux accélérateurs, mais de donner l'illusion aux applications qu'elles ont la capacité d'utiliser un nombre d'accélérateurs qui n'est pas lié au nombre d'accélérateurs physiques disponibles dans le matériel. Vu que ces accélérateurs utilisables par les applications n'ont pas d'existence physique, nous avons choisi de les appeler des « accélérateurs virtuels ».

2. `www.virtualbox.org`
3. `www.vmware.com`

4.2.2.1 Modèle de programmation

Nous proposons ici un modèle de programmation, basé sur MPI, qui a la propriété d'autoriser le programmeur à utiliser toutes les ressources de la grappe hybride. Ce modèle consiste ainsi à fournir à chacune des tâches MPI d'une application un accélérateur virtuel. Nous arrivons donc à une notion de virtualisation où les tâches MPI n'accèdent plus directement à un accélérateur physique mais à un accélérateur virtuel.

L'approche que nous avons choisie est de donner l'illusion à chaque tâche MPI qu'elle possède son propre accélérateur. La figure 4.1 montre un exemple où une application utilise quatre accélérateurs virtuels alors qu'il n'en existe que deux physiques (noté *Acc*). Ainsi, plutôt que d'utiliser des tâches MPI hybrides et des tâches uniquement CPU, nous pouvons autoriser la création de tâches toutes hybrides, et donc toutes identiques. Ainsi, le programmeur peut réutiliser le partitionnement effectué lors de la conception de son programme pour les architectures homogènes : au lieu de considérer que chaque processeur a une puissance identique, il peut considérer que chaque couple {processeur + accélérateur virtuel} a une puissance identique.

Ce modèle est particulièrement adapté aux applications hybrides SPMD (utilisant MPI) qui comportent des calculs s'exécutant sur les CPU et d'autres s'exécutant sur les GPU. Chaque tâche MPI peut ainsi utiliser les CPU pour certains calculs, et chaque tâche utilise également ses accélérateurs virtuels lorsque des calculs accélérés sont lancés. La mise en place des accélérateurs virtuels, à partir des accélérateurs physiques, doit faire l'objet d'une attention particulière afin d'avoir de bonnes performances : c'est l'objet de la section suivante.

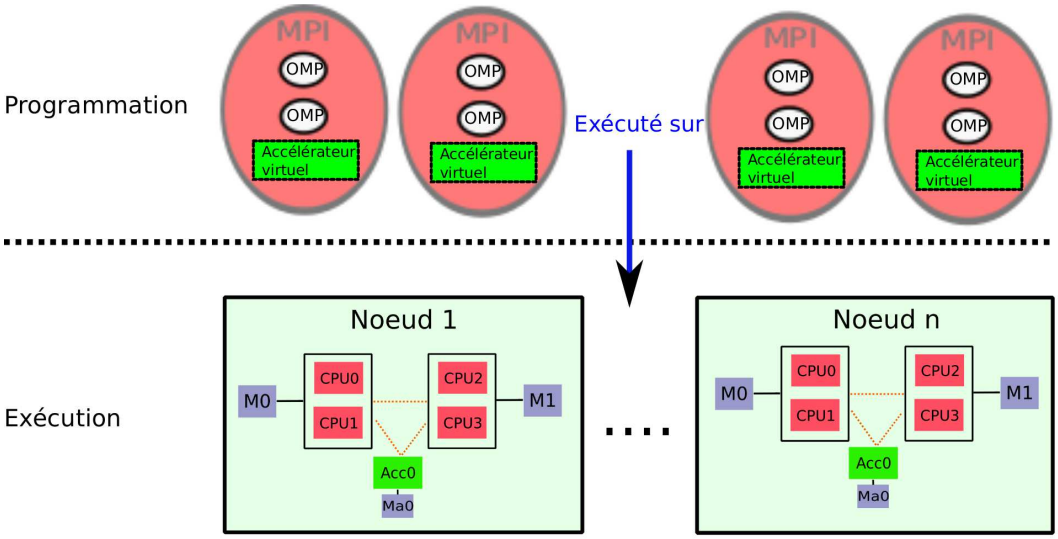
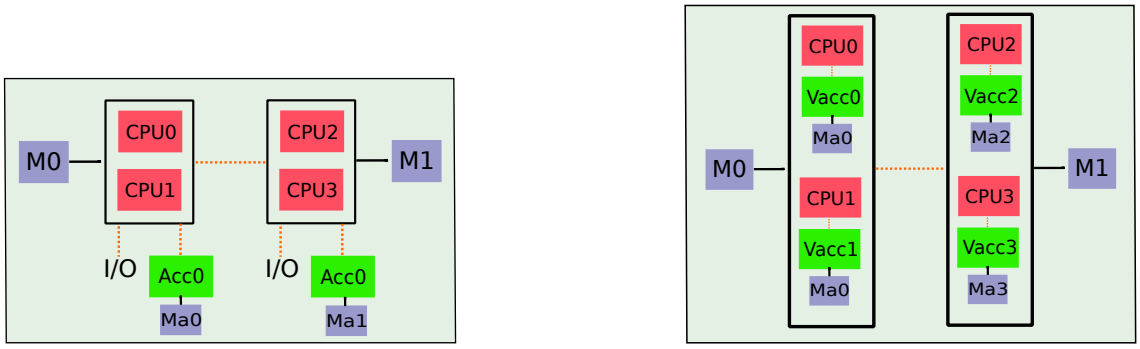


Figure 4.1 – Modèle de programmation : les quatre tâches MPI utilisent toutes un accélérateur virtuel.

4.2.2.2 Modèle d'exécution : partage des accélérateurs

Le modèle de machine sur lequel nous avons travaillé consiste en un ensemble de cœurs CPU et d'accélérateurs, détaillé sur la figure 4.2a. Le modèle de machine sur laquelle s'exécute le modèle de programmation est représentée sur la figure 4.2b. Ce modèle consiste à avoir un accélérateur virtuel (noté *Vacc*) par tâche MPI.



(a) Modèle de machine physique

(b) Modèle de machine virtualisée

Figure 4.2 – Modèle de machine hybride virtualisée.

Pour rendre disponible ces accélérateurs virtualisés, notre approche consiste à partager un accélérateur physique parmi les tâches qui utilisent les accélérateurs virtuels. Cette approche est représentée sur la figure 4.3 : quatre tâches voient un accélérateur ; il existe quatre accélérateurs virtuels, mais un seul est physique. Les quatre tâches partagent donc un unique accélérateur.

La question centrale de cette approche est le moyen choisi pour mettre en place le partage des accélérateurs. En fonction des caractéristiques des accélérateurs, le partage doit utiliser un ordonnancement adapté. Par exemple, si un accélérateur accepte simultanément un calcul et un transfert, il est possible d'ordonnancer le calcul d'une tâche avec le transfert d'une autre. L'ordonnancement mis en place est FIFO, les tâches MPI soumettent des requêtes, et ces requêtes sont exécutées sur les accélérateurs dans l'ordre de soumission. Vu que les tâches MPI travaillent sur des données indépendantes, il est possible d'ordonnancer un transfert soumis par une tâche MPI au même moment qu'un calcul soumis par une autre.

Nous considérons uniquement des architectures hybrides dont tous les nœuds sont hybrides et identiques : c'est le cas de la plupart des grappes actuelles. Nous pouvons ainsi créer des accélérateurs virtuels à partir des accélérateurs physiques de chaque nœud. Il n'y a donc pas de notion d'accès distant entre un accélérateur virtuel d'un nœud et un accélérateur physique d'un autre.

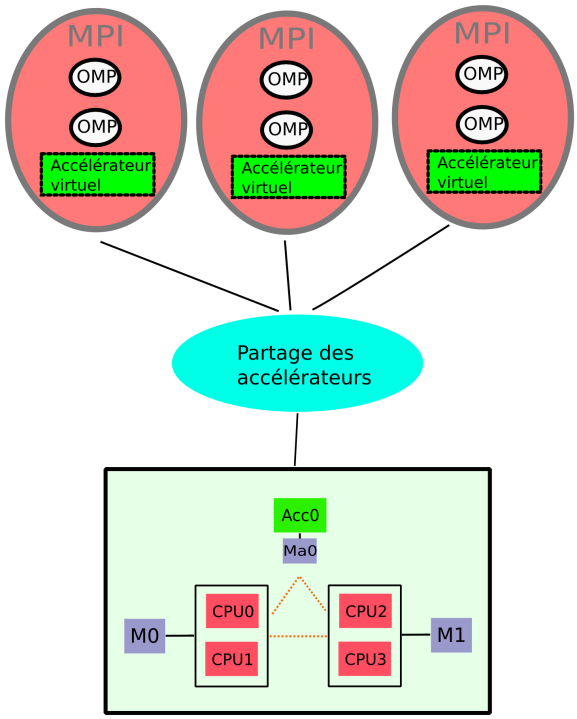


Figure 4.3 – Modèle d'exécution virtualisé

Utilisation CPU et accélérateurs Le modèle de programmation que nous proposons est synchrone : quand un accélérateur virtuel travaille, la tâche MPI est en attente d'un résultat en provenance de l'accélérateur. Ainsi, au sein d'une tâche MPI, il n'est pas possible d'utiliser simultanément CPU et accélérateur. Ce comportement est montré sur la figure 4.4.

Cependant, lorsque l'on considère l'ensemble des tâches MPI du nœud, quand certaines tâches MPI travaillent sur des CPU, d'autres peuvent utiliser les accélérateurs. Ainsi, sur la figure 4.5, deux tâches MPI sont lancées. Quand la première utilise une ressource CPU, la deuxième peut utiliser une ressource accélérateur. Il faut cependant noter que dans le cas d'un programme où les tâches sont fortement synchronisées entre elles – elles utilisent les accélérateurs toutes au même moment –, l'utilisation simultanée des accélérateurs et des CPU n'est plus possible.

4.2.2.3 Bilan et limite de l'approche

La virtualisation telle que nous l'avons présentée propose des avantages comme l'indépendance des programmes aux architectures ou la faible modification à apporter aux codes existants pour les exploiter. Elle est, de plus, parfaitement adaptée aux grands programmes hybrides fonctionnant sur des grappes qui utilisent une organisation SPMD avec

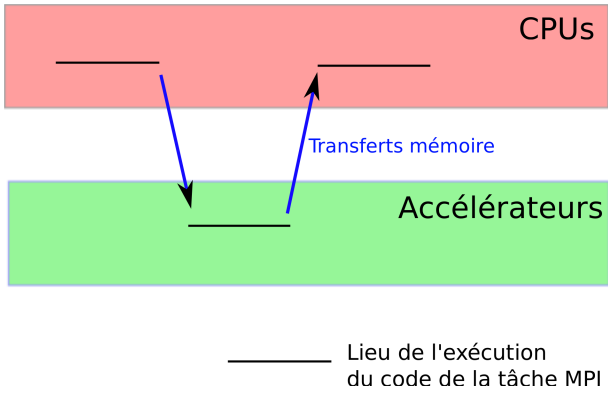


Figure 4.4 – Modèle de programmation synchrone

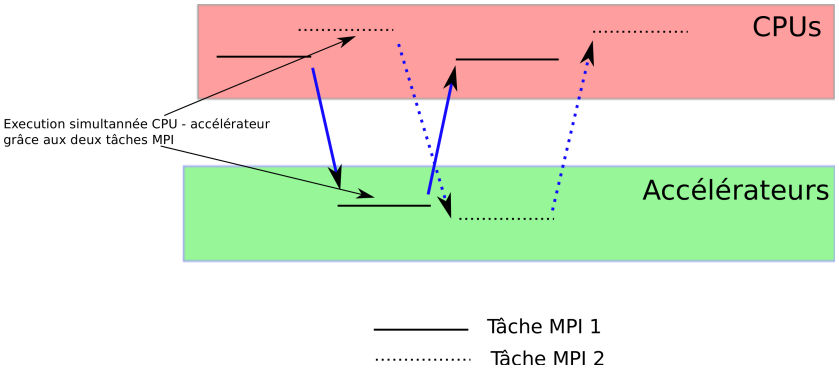


Figure 4.5 – Plusieurs tâches MPI permettent une utilisation simultanée CPU et accélérateur

l'utilisation, par exemple, de MPI. Cependant, l'emploi simultané des accélérateurs et des CPU est limité, et peut être inefficace si les tâches sont très fortement synchronisées (elles utilisent dans ce cas au même moment les CPU et les accélérateurs).

Nous explorons dans la suite de ce chapitre une évolution du modèle de programmation présenté ici, permettant de contrôler plus finement l'utilisation conjointe CPU et accélérateur.

4.2.3 Virtualisation des accélérateurs et des cœurs généralistes

Nous décrivons ici une extension du modèle précédent, basée sur l'objet abstrait {CPU + accélérateurs} nous permettant de faire des calculs hybrides. Nous commençons, dans cette partie, par définir précisément ce que nous entendons par calcul hybride, puis nous exposerons le modèle de programmation et d'exécution.

4.2.3.1 Modèle de programmation d'une parallélisation hybride

Nous considérons que l'exécution d'un calcul est hybride lorsque une partie de cette exécution utilise les CPU et l'autre les accélérateurs ; les unités de calcul sur lesquelles l'exécution a lieu ainsi que le code binaire des deux parties du calcul sont différents. Les espaces mémoire étant disjoints sur la grande majorité des architectures utilisant des accélérateurs, un calcul hybride doit donc s'exécuter également au sein de ces différents espaces mémoire.

Exécuter un même calcul sur deux architectures différentes nécessite de le paralléliser et de distribuer l'exécution sur des accélérateurs et un ensemble de cœurs CPU. Nous avons choisi d'utiliser un modèle *fork-join*, où différents fils d'exécution s'exécutent sur les CPU et d'autres sur les GPU. Le *fork* crée les différents fils et le *join* permet de bloquer l'exécution jusqu'à ce que tous les fils soient arrivés à leurs termes et donc que le calcul hybride soit fini. C'est un modèle largement utilisé, comme par exemple dans OpenMP, et habituellement employé sur des architectures à mémoire partagée et à cohérence de cache, où tous les fils d'exécution ont accès à une même zone mémoire cohérente.

Nous proposons d'étendre ce modèle pour permettre son utilisation sur des architectures hybrides où il existe différentes zones mémoire disjointes. Ainsi, dans notre modèle, en plus de créer les fils d'exécution, le *fork* va aussi configurer les différentes zones mémoire pour assurer une cohérence de ces différentes zones. Cette configuration est réalisée grâce à des mouvements de données entre les différents espaces mémoire. Symétriquement, le *join* finalise l'exécution des threads et réalise des mouvements de données pour fusionner les résultats CPU et accélérateur. Ce modèle est représenté sur la figure 4.6.

Pour préciser ce modèle de parallélisation hybride, nous allons tout d'abord détailler comment le *fork* crée les fils d'exécution et comment les mouvements de données y sont réalisés pour assurer la cohérence des zones mémoire. Nous verrons également comment nous avons traité le problème de l'équilibrage de charge entre les différentes ressources de calcul.

Un modèle basé sur des threads dans MPI Comme nous l'avons vu, la programmation hybride réunissant une approche MPI pour communiquer dans les grappes associée à une parallélisation de type thread (OpenMP) prend de plus en plus d'ampleur dans les

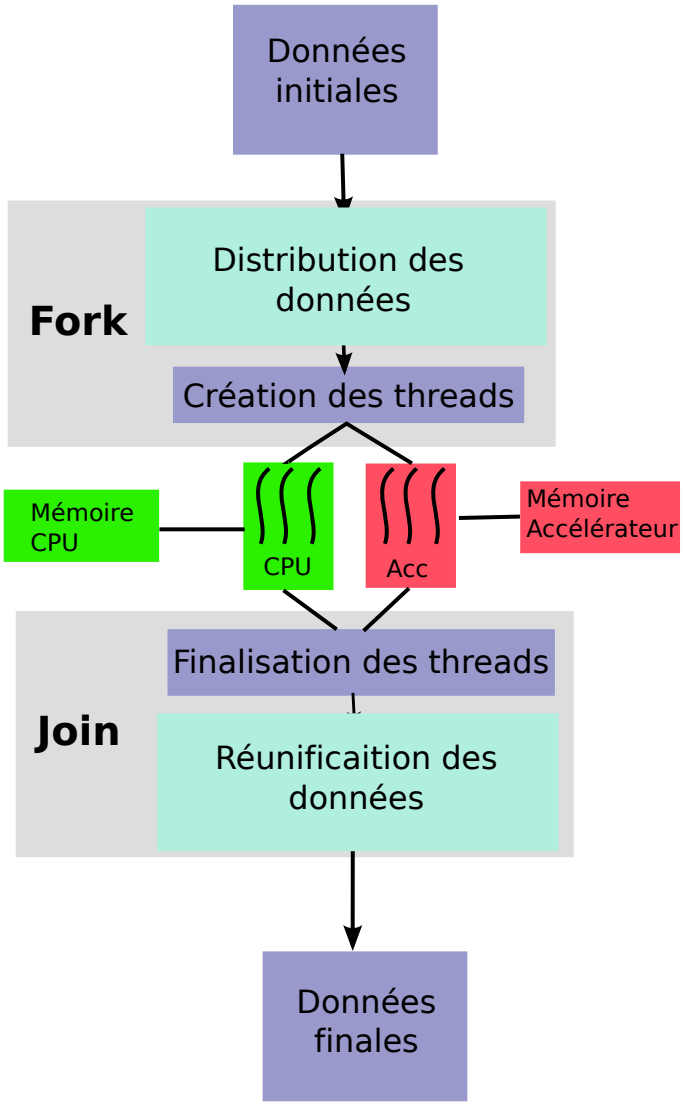


Figure 4.6 – Parallélisation hybride : mise en évidence des opérations fork et join.

```
1  identifiant_soumission =
2  soumettre_threads{
3                    1,4, //quatre CPU au maximum alloues
4                    1,1, //un accelerateur maximum alloue
5                    fonction_coupe(),
6                    fonction_regroupement(),
7                    fonction_multithreadee_cpu(),
8                    fonction_multithreadee_accelerateur(),
9                    }
```

Figure 4.7 – Déploiement des threads : demande de ressources et création des threads

codes de simulations. Nous proposons ici d'adapter ce modèle pour prendre en compte les grappes de calcul hybrides.

Le modèle de programmation proposé ici pour exprimer une parallélisation hybride se base donc sur une architecture bien établie où un programme parallèle est organisé en un ensemble de processus et de threads. De manière classique, un processus MPI peut créer des threads (avec OpenMP par exemple) pour ainsi paralléliser son exécution et exploiter les nœuds multi-cœurs ou multi-processeurs d'une grappe. Nous proposons ici d'étendre ce modèle et d'ajouter un second type de threads appelé "threads accélérateur" qui, plutôt que de s'exécuter sur des cœurs CPU, s'exécute sur des accélérateurs. Deux types de threads sont ainsi définis :

- **Threads CPU**, qui sont des threads POSIX standards ou des sections parallèles OpenMP. Ces threads contiennent du code exécutable sur des cœurs CPU.
- **Threads Accélérateur**, qui sont des threads contenant une succession de *kernel* CUDA ou OpenCL. Un thread accélérateur contient des appels à un nombre quelconque de kernels CUDA ou OpenCL.

Ainsi, la programmation CPU et accélérateur se fait grâce à des threads ; lors du *fork*, un processus peut créer des threads CPU et des threads accélérateur pour paralléliser son exécution de manière hybride. Ce processus pourra ainsi utiliser un ensemble de cœurs CPU et un ensemble d'accélérateurs pour réaliser un calcul hybride.

Déploiement des threads dans un environnement hybride Il reste maintenant à spécifier comment déployer nos différents types de threads dans un environnement hybride. Dans notre modèle, le déploiement consiste à sélectionner un ensemble de ressources puis à créer des threads qui vont exploiter ces ressources. C'est la fonction `soumettre_threads`, illustrée dans l'exemple de code 4.7, qui permet au programmeur d'exprimer les deux opérations du déploiement, précisées ci-dessous :

Sélection des ressources Lors de l'appel de la fonction `soumettre_threads`, le nombre de ressources de calcul que les threads peuvent utiliser est indiqué. Un nombre maximum($threads_{max}$) et minimum ($threads_{min}$) de ressources doit y être associé. En d'autres mots, $threads_{min}$ et $threads_{max}$ correspondent au minimum et au maximum de ressources de calcul utilisables par les threads. Pour les soumissions qui nécessitent plusieurs types de threads (CPU et accélérateur), ces informations doivent être spécifiées pour chaque type de ressources. Nous avons représenté, sur

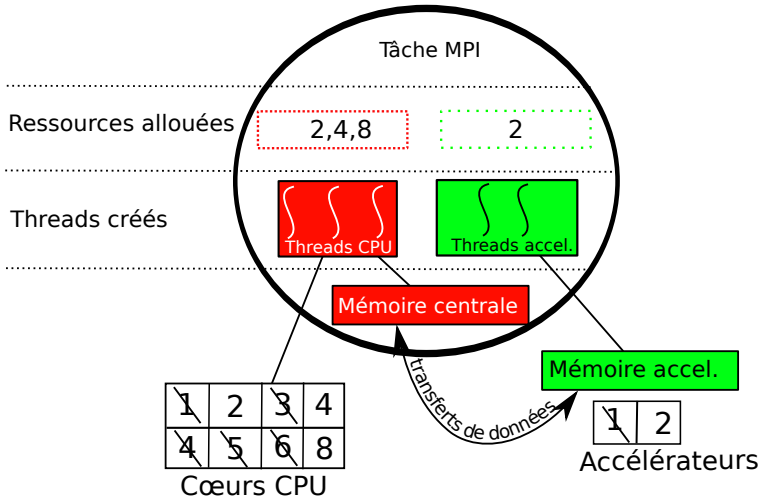


Figure 4.8 – Allocation des ressources et création des threads

la figure 4.7, un exemple de code où le programmeur demande entre 1 et 4 cœurs CPU (ligne 3) et 1 accélérateur (ligne 4).

En fonction du nombre de ressources disponibles sur le système (cœurs CPU ou accélérateurs), un allocateur de ressources a pour rôle de choisir un nombre de ressources compris entre $threads_{min}$ et $threads_{max}$. Cet allocateur de ressources est mis en œuvre dans un support exécutif et détaillé plus loin dans ce chapitre.

Création des threads Une fois les ressources allouées, les threads peuvent être créés. Lors de l'appel de `soumettre_threads`, la création des threads est encapsulée dans des fonctions fournies par le programmeur : il y a une fonction par type de ressources. Dans notre exemple figure 4.7, une fonction CPU (`fonction_multithreadee_cpu`, ligne 7) et une fonction accélérateur (`fonction_multithreadee_accelerateur`, ligne 8) sont passées en argument de `soumettre_threads`. Ces fonctions, exécutées par le support exécutif, peuvent créer un nombre de threads dépendant du nombre de ressources allouées (par exemple un thread accélérateur par accélérateur alloué).

La figure 4.8 illustre ce fonctionnement : trois ressources CPU – les processeurs 2, 4 et 8 – et l'accélérateur 2 ont été alloués (les processeurs 1, 3, 4, 5, 6 et l'accélérateur 1 étaient occupés et n'ont donc pas été alloués). Trois threads CPU et deux threads accélérateur sont créés pour utiliser ces ressources. Ici, les trois threads CPU pourront chacun utiliser un CPU alors que les deux threads accélérateur devront se partager un accélérateur physique. L'ordonnancement des threads sur des ressources libres sera couvert lors de la présentation du modèle d'exécution.

La fonction `soumettre_threads` est asynchrone et renvoie un identifiant. Cet identifiant peut être utilisé pour s'assurer de la terminaison des threads.

Nous avons vu la première partie du *fork* : la création de threads et la définition des ressources sur lesquelles ils s'exécutent. Il s'agit maintenant de détailler comment les mouvements de données doivent être effectués pour garantir une cohérence entre la mémoire des CPU et celle des accélérateurs.

Gestion des données

Présentation du problème

Dans le cas général où les accélérateurs et les processeurs centraux ont chacun leur propre mémoire, des mouvements de données doivent être prévus pour assurer la validité des calculs. Exécuter un calcul sur deux espaces mémoire disjoints implique de prêter attention à la version des données utilisée par les différentes ressources de calcul. En effet, la partie du calcul s'effectuant sur les accélérateurs et la partie travaillant sur les CPU doivent avoir accès à la même version des données. Ces données peuvent être, soit dupliquées, soit découpées :

— Données découpées : CPU et accélérateur ont chacun une partie de la donnée. Cette approche fonctionne bien sur des données simples comme des vecteurs, avec lesquelles le traitement peut être facilement découpé entre les zones mémoire. L'avantage est de réduire les temps de transferts entre les différentes zones mémoire puisqu'on ne transfère qu'une partie des données.

— Données dupliquées : CPU et accélérateur ont exactement les mêmes données mais les calculs CPU et accélérateur peuvent traiter des zones de mémoire différentes. Cette approche est intéressante lorsque l'on utilise des données complexes difficiles à découper. Le temps de transfert est cependant plus long par rapport au cas précédent.

A la fin du calcul, les données CPU et accélérateur ont toutes les deux été modifiées de manière indépendante. Dans notre modèle, il convient de fusionner les résultats des différents espaces mémoire pour prendre en compte les résultats provenant de toutes les ressources de calcul. Le résultat final peut être dans la mémoire CPU, dans la mémoire accélérateur, ou dans les deux. Ainsi, avant le démarrage d'un calcul hybride, des mouvements de données doivent êtres mis en place pour assurer la localisation correcte des données dans la mémoire des accélérateurs et dans la mémoire centrale. De même, à la fin d'un calcul hybride, d'autres mouvements de données sont nécessaires pour réunifier les parties CPU et accélérateur du calcul.

Mise en œuvre

Dans notre modèle, lorsque les threads sont créés, ils doivent avoir accès aux bonnes versions des données. Il convient donc de lancer les mouvements de données avant le démarrage des threads. Sur l'exemple de la figure 4.8, les deux threads accélérateur et les trois threads CPU doivent avoir accès à des données correctes sur les ressources allouées et des mouvements de données doivent donc être initiés entre différentes zones mémoire. Ainsi, notre modèle de programmation hybride définit deux fonctions devant être fournies par le programmeur lors de l'appel de `soumettre_threads`. La première, appelée "fonction de coupe" a pour but de mettre en place les mouvements de données nécessaires au calcul hybride. Inversement, une "fonction de regroupement" (ligne 6 dans la figure 4.7) est destinée à réunir les données des différentes unités (CPU et accélérateurs) impliquées dans le calcul. La fonction de coupe est appelée avant la création des threads et permet donc de garantir que les données seront à jour pour l'exécution des threads. La fonction de regroupement est quant à elle appelée juste après la fin de l'exécution des threads.

Pour donner un exemple, la figure 4.9 représente une fonction de coupe qui découpe un vecteur afin qu'une partie du traitement puisse s'exécuter sur des CPU et un accélérateur. Cette fonction de coupe va envoyer la partie gauche du vecteur vers l'accélérateur.

```
1 int donnees[1..100] // vecteur de 100 nombres entiers
2 fonction_coupe()
3 {
4     envoie_accelerateur(donnees[1..50])
5 }
```

Figure 4.9 – Fonction de coupe statique

Les CPU conservent la totalité du vecteur et pourront ainsi traiter uniquement sa partie droite. Le découpage présenté ici est statique ; le programmeur décide d'attribuer à l'accélérateur la partie gauche du vecteur. Nous verrons dans la suite qu'une découpe dynamique est plus appropriée lorsque l'on parle d'architectures hybrides et de ressources de calcul hétérogènes.

Équilibrage de charge hybride

Description du problème

Dans les architectures non homogènes que nous utilisons, la puissance de calcul de l'ensemble des CPU et celle des accélérateurs est différente. Par exemple, un processeur récent atteint près de 100 Gflops alors qu'un accélérateur graphique NVIDIA TESLA C2090 se rapproche du Tflops. La figure 4.11a montre une exécution parallèle hybride où le calcul effectué sur le CPU dure beaucoup plus longtemps que celui exécuté sur l'accélérateur. L'accélérateur se met donc en attente jusqu'à la fin de l'exécution du calcul CPU. Pour éviter ce problème, un équilibrage de charge entre les deux unités de calcul pourra minimiser le temps d'attente : les temps d'exécution sur les accélérateurs et sur les CPU doivent être les plus proches possible.

Afin de permettre aux applications de s'adapter aux différents types de machines, machines pour lesquelles la puissance de calcul des accélérateurs et des cœurs CPU est différente, nous voulons offrir un équilibrage de charge dynamique, c'est-à-dire qui puisse s'adapter à la puissance de la machine hybride utilisée. Dans notre modèle, l'équilibrage de charge peut être réalisé de deux façons :

- Équilibrage en répartissant les données : le temps d'exécution d'un calcul est fonction de la quantité de données traitée par celui-ci. En se basant sur cette propriété, attribuer plus de données à un calcul augmentera, dans la plupart des cas, son temps d'exécution et permettra donc de répartir la charge.
- Équilibrage par allocation des ressources de calcul. Le temps d'exécution d'un programme parallèle est fonction du nombre de ressources de calcul qu'il utilise. Ainsi, attribuer plus ou moins de ressources de calcul à la partie CPU ou accélérateur d'un calcul permet, en général, de répartir la charge.

Nous nous sommes concentrés dans nos travaux sur l'équilibrage dirigé par les données.

Mise en place de l'équilibrage de charge

Nous avons choisi d'utiliser une méthode, basée sur un historique, qui permet de prédire une répartition de données destinée à équilibrer la charge entre les ressources matérielles impliquées dans un calcul. Notre méthode sera détaillée dans la partie traitant du modèle d'exécution.

Dans notre modèle de programmation, c'est la fonction de coupe qui permet d'exprimer

la mise en place de mécanismes destinés à équilibrer la charge ; l'approche que nous avons utilisée consiste à ajouter un argument à cette fonction (`fonction_coupe(int repartition)`. L'argument `repartition` représente le pourcentage de données que les CPU doivent traiter et est déterminé grâce à notre méthode de calcul de la répartition. Une répartition de 0 signifie que le CPU traite toutes les données du calcul et une répartition de 100 signifie que toutes les données sont traitées par les accélérateurs. Une valeur intermédiaire ($0 < repartition < 100$) permet donc de répartir la charge de calcul sur les différentes ressources matérielles. La fonction de coupe, fournie par le programmeur, doit donc prendre en compte l'argument `répartition` et l'utiliser pour distribuer les données du calcul sur les différentes ressources matérielles qui ont été allouées.

En adaptant l'exemple de la figure 4.9, on définit une nouvelle fonction de coupe qui prend maintenant l'argument supplémentaire `répartition` et l'utilise pour découper les données (figure 4.10).

```
1 int donnees[1..100] //vecteur de 100 nombres entiers
2 fonction_coupe(int repartition)
3 {
4   envoie_accelerateur(donnees[1..repartition])
5 }
```

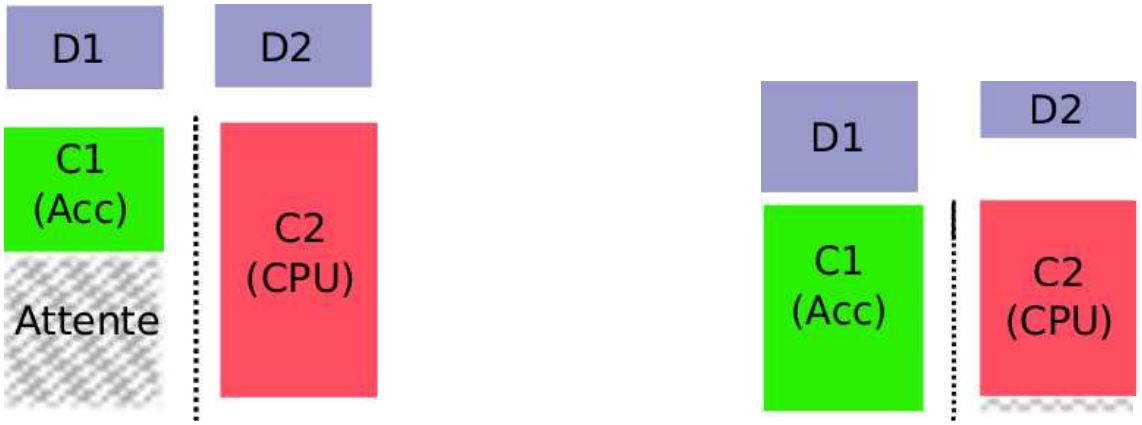
Figure 4.10 – Fonction de coupe dynamique

4.2.3.2 Exemple d'un programme hybride

Pour illustrer, nous allons prendre un exemple d'un code (figure 4.12) qui réalise un produit scalaire entre deux vecteurs. Nous considérons, dans un premier temps, un programme MPI dont une tâche "maître" distribue les données sur plusieurs tâches esclaves. Les tâches esclaves utilisent une construction OpenMP pour faire les calculs de manière multithreadées (ligne 26 à 30). Pour fixer les idées, nous considérons un programme qui crée deux tâches esclaves.

Nous proposons maintenant d'étendre ce programme MPI en utilisant notre modèle de programmation afin d'exploiter des ressources CPU et accélérateur simultanément. Le programme adapté est représenté sur la figure 4.13 et nous proposons de détailler ici les étapes permettant d'aboutir à ce programme. Nous avons besoin de deux fonctions, responsables de la création des threads CPU et accélérateur. La première, qui crée les threads CPU, reprend une portion de code adaptée du programme MPI figure 4.12 ; il s'agit d'encapsuler le code OpenMP dans la fonction `fonction_cpu` :

```
//start et stop définissent la portion des données à traiter
fonction_cpu(int *données_reçuesA, int *données_reçuesB, int start,int stop)
{
   #pragma omp parallel for reduction(+:somme)
   for(i=start;i<=stop;++i)
   {
     somme += données_reçuesA[i]*données_reçuesB[i]
```



(a) Mauvais équilibrage de charge : le calcul accélérateur attend

(b) Équilibrage réalisé : le temps d'attente est minimum et le temps d'exécution global diminue

Figure 4.11 – Équilibrage de charge d'une parallélisation hybride.

```
   }
}
```

La seconde fonction (`fonction_accélérateur`) contient du code destiné aux accélérateurs. Il doit être équivalent au code CPU, à savoir, réaliser un produit scalaire entre deux vecteurs. Pour plus de concision, nous ne représentons pas ici le code accélérateur qui peut être écrit, par exemple, en CUDA. Notre modèle de programmation définit des moyens de créer des threads (fonction `créer_thread`) et de les associer à du code accélérateur (fonction `associer_thread`).

```
fonction_accélérateur(int donnee,int start,int stop)
{
type_thread thread1;
créer_thread(thread1);
associer(thread1, code accélérateur,donnée,start,stop)
}
```

La fonction de coupe, qui distribue les données entre les threads crées par `fonction_accélérateur` et `fonction_cpu` peut s'écrire :

```
fonction_coupe(int répartition)
{
envoie_accélérateur(donnéesA[1..répartition])
envoie_accélérateur(donnéesB[1..répartition])
}
```

```
1  if(rang_mpi == rang_maitre)
2   {
3     int donneesA[1..200] // donnees initiales : vecteur d'entiers A
4     int donneesB[1..200] // donnees initiales : vecteur d'entiers B
5
6     MPI_Send(1,donneesA[1..100]) //envoie la moitie au rang 1
7     MPI_Send(1,donneesB[1..100]) //envoie la moitie au rang 1
8     MPI_Send(2,donneesA[100..200]) //envoie l'autre moitie au rang 2
9     MPI_Send(2,donneesB[100..200]) //envoie l'autre moitie au rang 2
10
11    int somme,res;
12    somme =0;
13    MPI_Reduce(&somme, &res , 1, MPI_INT, MPI_SUM,0)
14
15    afficher "produit␣scalaire␣:␣", somme;
16 }
17 else
18 {
19    int donnees_recuesA[1..100]
20    int donnees_recuesB[1..100]
21    MPI_Recv(donnees_recuesA[1..100])
22    MPI_Recv(donnees_recuesB[1..100])
23
24    int somme = 0;
25    int c;
26    #pragma omp parallel for reduction(+:somme)
27    for(i=1;i<=100;++i)
28    {
29      somme += donnees_recuesA[i]*donnees_recuesB[i]
30    }
31
32
33   MPI_Reduce(&somme, &c, 1, MPI_INT, MPI_SUM,0)
34 }
35 }
```

Figure 4.12 – Exemple de code OpemMP et MPI calculant un produit scalaire

Une fonction de regroupement, qui récupère du côté accélérateur la valeur calculée peut s'écrire de la manière suivante :

```
fonction_regrouppement(int répartition)
{
sommeAcc = recoit_accélérateur(valeur_acc); //on lit la valeur
                                        //calculee par l'accélérateur

somme_hybride = sommeCPU + sommeAcc; //on construit un resultat prenant
                              //en compte le calcul accelérateur et  CPU
}
```

Dans nos exemples, nous considérons que la fonction `soumettre_threads` prend simplement des données en arguments et que ces données sont disponibles pour les fonctions de coupe/regroupement. Nous verrons plus en détail, dans le chapitre suivant, comment ces données sont passées à ces deux fonctions.

Nous représentons, dans la figure 4.13, le code MPI adapté dans lequel les tâches esclaves utilisent notre modèle de programmation pour paralléliser leurs exécutions entre CPU et accélérateur. Dans ce code, le programmeur a choisi d'utiliser entre 1 et 4 CPU et 1 accélérateur (fonction `soumettre_threads` lignes 30-38). Il a également précisé dans cette fonction qu'il souhaitait utiliser la fonction de coupe (ligne 33) et la fonction de regroupement (ligne 34) que nous avons décrites auparavant.

La fonction `attendre()` est bloquante jusqu'à ce que tous les threads soient terminés et permet ainsi de s'assurer de la fin d'un calcul hybride.

En résumé, pour effectuer une parallélisation hybride dans le modèle de programmation présenté dans cette partie, le programmeur doit fournir les trois objets suivants :

1. Des threads CPU et accélérateur pour effectuer la parallélisation hybride. Ces threads sont exécutés sur des ressources allouées par notre support exécutif.
2. Une fonction de coupe, destinée à mettre en place la mémoire centrale et la mémoire des accélérateurs. Cette fonction prend le paramètre `répartition` en argument.
3. Une fonction de regroupement dont le but est de réunir les contributions des cœurs CPU et des accélérateurs.

4.2.3.3 Modèle d'exécution

Le modèle de programmation qui a été présenté nécessite un support exécutif capable de sélectionner un ensemble de ressources, de réguler la charge de calcul entre les différentes ressources matérielle et d'exécuter des threads sur les ressources allouées.

Allocation des ressources Lorsque les threads sont soumis avec la fonction `soumettre_threads`, l'allocateur a le rôle de fixer le nombre de ressources utilisables pour l'exécution des threads. Pour les soumissions qui demandent uniquement des CPU ou des accélérateurs

```
1  if(rang_mpi == rang_maitre)
2   {
3      int donneesA[1..200] // donnees initiales : vecteur d'entiers A
4      int donneesB[1..200] // donnees initiales : vecteur d'entiers B
5
6      MPI_Send(1,donneesA[1..100]) //envoie la moitie au rang 1
7      MPI_Send(1,donneesB[1..100]) //envoie la moitie au rang 1
8      MPI_Send(2,donneesA[100..200]) //envoie l'autre moitie au rang 2
9      MPI_Send(2,donneesB[100..200]) //envoie l'autre moitie au rang 2
10
11     int somme,res;
12     somme =0;
13     MPI_Reduce(&somme, &res , 1, MPI_INT, MPI_SUM,0)
14
15     afficher "produit␣scalaire␣:␣", res;
16 }
17 else
18 {
19     int somme = 0;
20     int c;
21     int identifiant_soumission;
22
23     int donnees_recuesA[1..100]
24     int donnees_recuesB[1..100]
25     MPI_Recv(donnees_recuesA[1..100])
26     MPI_Recv(donnees_recuesB[1..100])
27
28
29     identifiant_soumission =
30     soumettre_threads{
31                       1,4, //quatre CPU au maximum alloues
32                       1,1, //un accelerateur maximum alloue
33                       fonction_coupe(),
34                       fonction_regroupement(),
35                       fonction_cpu(),
36                       fonction_accelerateur(),
37                       donnees_recuesA[1..100],donnees_recuesB[1..100],
                             somme_hybride
38                      }
39
40     attendre(identifiant_soumission);
41
42
43     MPI_Reduce(&somme_hybride, &c, 1, MPI_INT, MPI_SUM,0)
44 }
```

Figure 4.13 – Exemple de code MPI utilisant des threads CPU et accélérateur

(soumissions homogènes), il s'agit de déterminer le nombre de ressources à allouer aux CPU ou aux accélérateurs. Pour les soumissions hybrides, c'est-à-dire les soumissions qui demandent des ressources CPU et accélérateur, il s'agit en plus de déterminer une répartition de la charge de calcul entre les CPU et les accélérateurs. Les soumissions des threads sont stockées dans trois queues décrites ci-dessous :

— Une "queue CPU" qui contient des soumissions de threads n'utilisant que les CPU
— Une "queue accélérateur" qui contient des soumissions de threads n'utilisant que les accélérateurs
— Une "queue hybride" qui contient des soumissions hybrides. Cette queue contient donc des soumissions de threads accélérateur et CPU.

L'allocation des ressources est décrite en détail sur la procédure 1. L'algorithme scrute successivement le contenu des trois queues pour déterminer un ensemble de ressources à allouer à chaque soumission. Nous distinguons deux cas selon que la soumission est hybride ou homogène.

– **Soumissions homogènes** Pour ce type de soumission, l'allocateur va allouer un nombre de ressources le plus proche du $threads_{max}$ indiqué par le programmeur lors de la soumission. Ces ressources sont toutes du même type (CPU ou accélérateur). Le calcul de cette quantité de ressources est décrite sur la procédure 2 ; le nombre de ressources allouées correspond au maximum entre le nombre de ressources demandées et le nombre de ressources libres sur la machine.

  Une fois les ressources allouées, les threads peuvent y être exécutés.

– **Soumissions hybrides** L'allocation est similaire aux soumissions homogènes à la différence que ce type de soumission demandant deux type de ressources, il y a une allocation de cœurs CPU et une allocation d'accélérateurs. La procédure 2 est appelée deux fois : la première fois pour la partie CPU de la soumission, la seconde pour la partie accélérateur.

  Contrairement aux soumissions homogènes, une fois que l'allocation a eu lieu, les threads ne démarrent pas immédiatement, en effet, l'étape qui consiste à distribuer et réguler la charge entre CPU et accélérateur a alors lieu.

Distribution et régulation de la charge Pour les soumission hybrides, une répartition de la quantité de données à traiter pour les CPU et les accélérateurs doit donc être calculée. Nous allons reprendre notre exemple du produit scalaire et montrer comment nous avons choisi de mettre en œuvre ce calcul de répartition.

Notre stratégie d'équilibrage est prévue pour fonctionner avec des applications scientifiques qui sont bien souvent basées sur des algorithmes itératifs, où chaque itération exécute des mêmes noyaux de calcul. Nous allons donc ici supposer que le calcul du produit scalaire est appelé pour plusieurs itérations.

La figure 4.14 montre le fonctionnement général de notre méthode d'équilibrage de charge. Elle est implémentée dans la fonction *CalculRépartition* sur la procédure 1. L'idée est de commencer un calcul en répartissant les données du problème équitablement sur les différentes ressources (Itération 1 sur la figure 4.14). La partie CPU et accélérateur du produit scalaire traitent donc la même quantité de données. Les temps d'exécution de la partie CPU et accélérateur sont enregistrés et sont utilisés pour construire deux fonctions

Procédure 1 Allocation des ressources

Entrée : $Queue\_CPU$, $Queue\_Accélérateur$ and $Queue\_Hybride$
Sortie : Une allocation de ressources

```
loop
  if Queue_CPU.non_vide then
    El_CPU ← fin de la queue CPU
    getRessources(El_CPU, ressources_disponibles)
    if ressources_disponibles ≠ 0 then
      {allouer ressources_disponibles coeurs CPU}
      {Démarrage des threads}
    end if
  end if
  if Queue_Accélérateur.non_vide then
    {...Comme la partie CPU ...}
  end if
  if Queue_Hybride.non_vide then
    Element_Hybride ← fin de la queue hybride
    getRessources(Element_Hybride.El_CPU, ressources_disponibles_CPU)
    getRessources(Element_Hybride.El_Accélérateur, ressources_disponibles_Accélérateur)

    if ressources_disponibles_CPU + ressources_disponibles_Accélérateur ≠ 0 then
      if ressources_disponibles_CPU = 0 then
        répartition ← 0%
      else if ressources_disponibles_Accélérateur = 0 then
        répartition ← 100%
      else
        répartition ← CalculRépartition(ressources_disponibles_CPU,
        ressources_disponibles_Accélérateur, Element_Hybride.id)
      end if
      {allouer ressources_disponibles_CPU coeurs CPU}
      {allouer ressources_disponibles_Accélérateur accélérateurs}
      {exécuter fonction_de_coupe(répartition)}
      {démarrer les threads CPU et accélérateur}
      {exécuter fonction_de_regroupement(répartition)}
    end if
  end if
end loop
```

Procédure 2 getRessources, recherche le maximum de ressources utilisable pour une soumission de threads

Entrée : Un objet *ressources*, qui gère les ressources de calcul libres ou occupées et une soumission *El*

Sortie : Une liste de ressource maximum utilisable par la soumission

```
n ← max(ressources.libre, El.max)
if n ≤ El.min then
  ressources_disponibles ← 0
else
  ressources_disponibles ← n
end if
```

mathématiques d'approximation qui mettent en relation le temps d'exécution et le volume de données traité par les ressources impliquées dans le calcul. Ces deux fonctions, que nous appelons *fonction de coût accélérateur* et *fonction de coût CPU* sont représentées sur la figure 4.14 par deux graphiques. Lorsque le même noyau de calcul est exécuté une seconde fois, la répartition est cette fois-ci fixée à 70% / 30%, et un nouveau point est ajouté aux deux fonctions (Itération 2). A partir de ce moment, il est possible d'approximer l'expression des fonctions en utilisant des algorithmes d'interpolation. Nous avons choisi d'utiliser l'algorithme de Neville [PTVF07] qui est un algorithme d'interpolation polynomiale.

Considérons que l'on a $n+1$ points sur chacune de nos courbes. L'algorithme est basé sur la propriété décrite par l'équation (4.1)

$$p_{i,j+1}(x) = \frac{(x_i - x)p_{i+1,j}(x) + (x - x_{i+j+1})p_{i,j}(x)}{x_i - x_{i+j+1}} \tag{4.1}$$

avec :

$$p_{i,0}(x) = y_i \tag{4.2}$$

$p_{i,m}$ correspond au polynôme interpolé entre les points i et m. D'après l'équation (4.2), il est possible de calculer simplement les $p_{i,0}(x)$ pour $i \in [0..n]$. ($p_{i,0}(x)$ est un polynôme de degré 0 qui correspond à la valeur du point i). En utilisant les $p_{i,0}(x)$ dans la relation de récurrence (4.1) on peut calculer le polynôme $p_{0,n}(x)$ qui est une approximation de la fonction passant par les $n+1$ points mesurés.

Nous avons donc maintenant deux fonctions qui sont exprimées sous forme de polynôme ; il est donc possible d'extrapoler des points. Il suffit ainsi de déterminer un point où ces deux fonctions sont égales pour avoir une répartition de données qui équilibre la charge entre les deux ressources de calcul.

Dans notre exemple du produit scalaire, pour l'itération 3, la recherche d'un point où les deux fonctions interpolées sont identiques permet de déterminer que pour 20% des données sur le CPU et 80% sur l'accélérateur, les temps d'exécution CPU et accélérateur sont proches : c'est cette répartition qui est attribuée au calcul du produit scalaire à l'itération 3. Il faut noter que les points estimés peuvent donner des résultats éloignés de la réalité, ainsi, pour affiner encore la prédiction, les temps d'exécution effectifs des points estimés sont ajoutés à nos fonctions de coût et sont utilisés pour les prédictions futures.

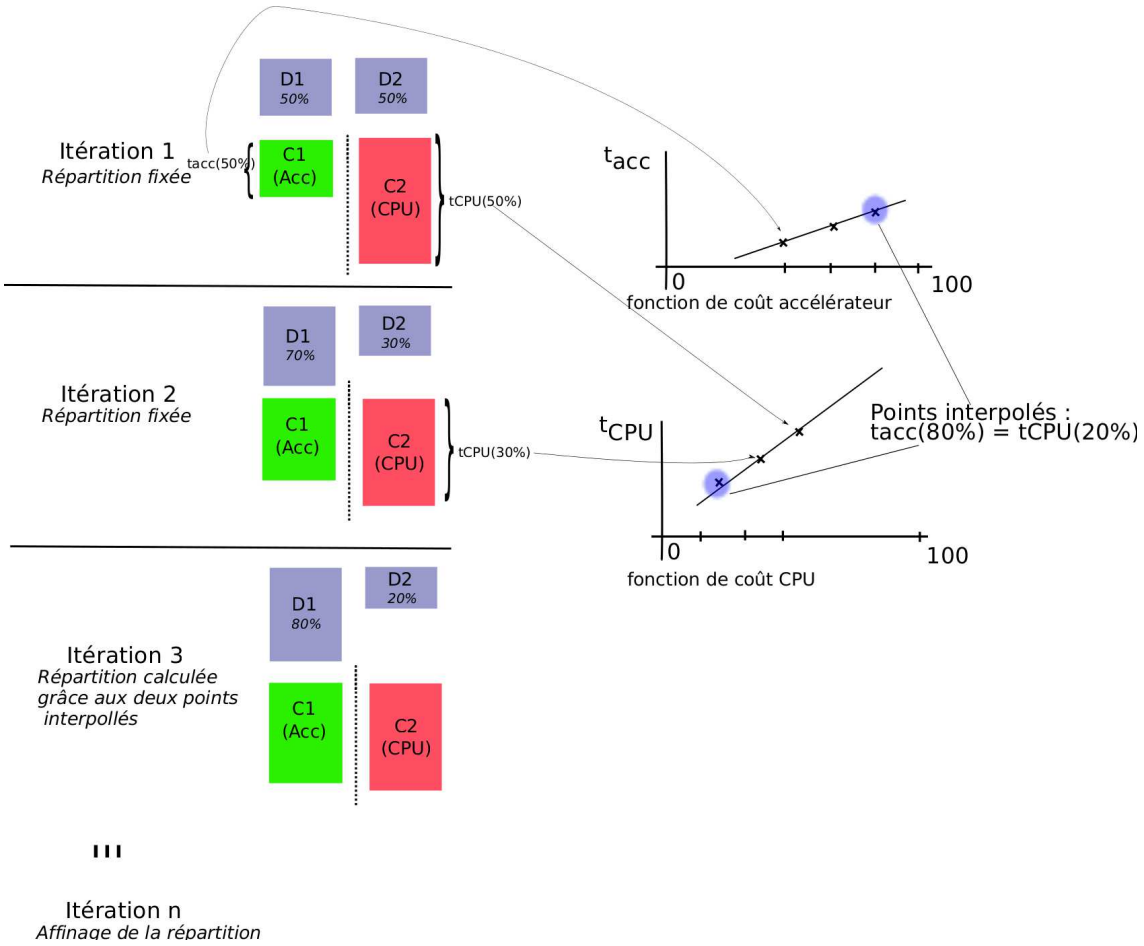


Figure 4.14 – Illustration de notre méthode d'équilibrage de charge sur plusieurs exécutions d'un produit scalaire

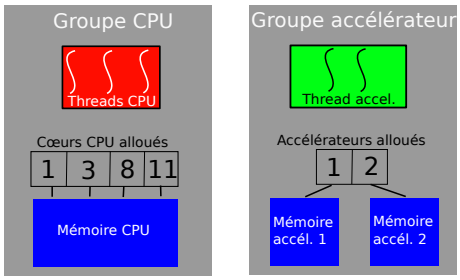


Figure 4.15 – Groupe de threads CPU et accélérateur

Une fois la charge de calcul déterminée, la fonction de coupe est exécutée afin de mettre en place les données sur les ressources hybrides allouées. Enfin, les threads CPU et accélérateur sont prêts à être démarrés.

Ordonnancement des threads Les threads ne sont pas manipulés seuls mais à l'intérieur de "groupes de threads" illustrés sur la figure 4.15. Le groupe de threads est un ensemble de threads qui s'exécute sur des ressources du même type associé à des informations concernant les ressources matérielles sur lesquelles ces threads vont s'exécuter. Il y a donc deux types de groupes de threads : les groupes CPU et les groupes accélérateur. Nous avons choisi d'utiliser des groupes de threads afin que l'ordonnanceur sache exactement sur quelles ressources les threads doivent être placés.

Nous considérons que les threads CPU s'exécutent sur des machines à mémoire cohérente et donc que chaque thread d'un groupe CPU a accès à une même zone mémoire cohérente. Pour les threads d'un groupe accélérateur, les zones mémoire des accélérateurs sont disjointes et des threads peuvent demander à être ordonancés sur un même accélérateur physique pour pouvoir partager simplement de la mémoire avec un autre thread. Nous détaillons maintenant comment nous avons choisi d'ordonnancer les threads de ces deux groupes.

– **Ordonnancement des threads CPU** La plupart des systèmes d'exploitation définissent une notion de threads CPU et sont capables de les ordonnancer sur des cœurs CPU. Nous avons donc confié l'ordonancement des threads CPU au système d'exploitation. Puisque les threads CPU ont accès à une même zone mémoire cohérente, nous avons choisi de fixer l'exécution des threads du groupe CPU sur l'ensemble des cœurs CPU alloués. Ainsi, chaque thread peut utiliser un cœur CPU parmi ceux alloués et c'est le système d'exploitation qui choisit où placer les threads.

– **Ordonnancement des threads accélérateur** Nous nous plaçons dans le contexte où un accélérateur est vu comme une "boîte noire" et où le contrôle du déroulement de l'exécution est confié à l'accélérateur. Dans notre cas, nous lançons un code exécutable sur un accélérateur et attendons sa fin. Contrairement aux CPU, ces accélérateurs n'ont pas de système d'exploitation permettant d'ordonnancer directement nos threads accélérateurs. Notre support exécutif réalise donc cet ordonnancement et place les threads accélérateur sur des ressources allouées

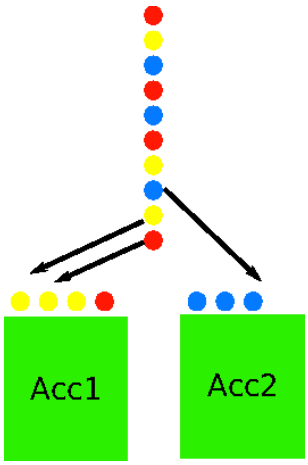


Figure 4.16 – Ordonnancement des threads accélérateur par couleur

Pour traiter le problème des zones mémoire non cohérentes, pour lequel différents threads souhaitant partager des informations nécessitent de s'exécuter sur un même accélérateur, nous avons choisi d'utiliser un mécanisme de coloriage. Ainsi, si plusieurs threads désirent s'exécuter sur le même accélérateur, ils le précisent lors de leurs créations et une couleur leur est attribuée. Ainsi, chaque thread de la même couleur s'exécute sur un même accélérateur. Sur des accélérateurs supportant une exécution concurante (certains processeurs GPU comme les NVIDIA FERMI ont cette possibilité), deux threads de la même couleur auront un accès simultané aux mêmes GPU. La figure 4.16 montre un exemple où deux accélérateurs traitent des threads de plusieurs couleurs. Lorsque un accélérateur traite un thread d'une couleur, tous les autres threads de cette couleur utiliseront cet accélérateur. Les threads qui ne sont pas coloriés n'ont pas de contraintes de placement et sont donc déployés sur un accélérateur alloué quelconque.

Pour réaliser des recouvrements entre d'éventuels transferts d'un thread avec les calculs d'un autre, nous avons utilisé la méthode décrite sur la figure 4.3 : il existe plusieurs accélérateurs virtuels pour un accélérateur physique. En fonction des caractéristiques de ces accélérateurs, les différents recouvrements possibles peuvent donc être réalisés. Pour l'exécution simultanée des CPU et des accélérateurs, celui-ci est naturel car l'ordonnancement de ces deux types de threads se fait de manière totalement désynchronisée.

4.3 Observation de programmes hybrides

Les modèles de programmation définis dans ce chapitre, les architectures hybrides, et les applications sont relativement complexes. Il n'est donc pas aisé de comprendre le déroulement d'une application s'exécutant sur une architecture hybride. Des informations précises, permettant de suivre le comportement d'un programme écrit avec nos modèles de programmation durant son exécution sont donc indispensables.

Notre objectif est de donner au programmeur des moyens pour qu'il comprenne le déroulement de son programme afin qu'il puisse analyser si les ressources de calcul ont bien été exploitées.

4.3.1 Que veut-on observer dans un programme hybride ?

Nous nous plaçons dans le contexte d'applications bâties autour d'un modèle SPMD et qui ont donc un comportement bien souvent identique sur tous les nœud de calcul d'une grappe hybride. Par conséquent, nous proposons de concentrer l'observation d'une application sur un nœud particulier des grappes de calcul. Dans le cas où l'application n'a pas le même comportement sur tous les nœuds, nous offrons également la possibilité d'observer le déroulement de l'application sur plusieurs nœuds.

Dans ce contexte, nous permettons l'observation de l'exécution de code sur les accélérateurs, les CPU, le poids des transferts mémoire et les différents recouvrements entre les calculs et les mouvements de données. L'équilibrage de charge entres les différentes ressources est importante pour les performances et nous proposons donc d'identifier à quel moment un cœur CPU et/ou un accélérateur est inoccupé pour déterminer si le calcul hybride utilise correctement les ressources qui lui sont allouées.

L'approche que nous avons retenue pour la visualisation est de l'effectuer après la fin de l'exécution du programme. Il s'agit donc d'une visualisation *post mortem*.

4.3.2 Traçage des applications

Nos modèles de programmation définissent différentes fonctions permettant d'initier des transferts, lancer des exécutions de code ou créer des threads. Notre support exécutif va ainsi enregistrer des informations à l'exécution de chacune de ces fonctions. Ainsi, pour chaque exécution d'un transfert mémoire ou d'un thread, des événements sont générés. Ces évènements sont estampillés avec les dates de début et de fin des exécutions et sont stockées dans un fichier sous la forme d'un fichier de trace OTF. Ainsi, aucun ajout de code spécifique de traçage n'est nécessaire dans les applications utilisant nos modèles de programmation et, par conséquent, l'intrusivité dans le code applicatif est nulle.

L'observation étant réalisée de manière *post mortem*, une fois que la trace OTF a été générée, cette trace peut est interprétée par un programme externe (dans notre cas le programme ViTE) afin d'afficher la trace ainsi obtenue de manière graphique.

4.3.3 Organisation de la visualisation

Nous avons choisi d'organiser la visualisation d'une application hybride en classant les différentes informations sous la forme d'un arbre. Chaque "nœud" de l'arbre correspond à un conteneur (figure 4.17). Les conteneurs sont insérés dans d'autres conteneurs et représentent chacun, soit une partie de la grappe de calcul hybride, soit une information relative à un calcul hybride.

Nous définissons quatre types de conteneurs principaux :

— **Le nœud**, qui représente un nœud hybride comportant des CPU et des accélérateurs. Le nombre de conteneurs nœud est égal au nombre de nœuds utilisés par l'application.

— **Le type d'unité de calcul**, contenue dans le conteneur "nœud", permet de visualiser le travail de chaque ressource de calcul (accélérateurs ou CPU). Il y a

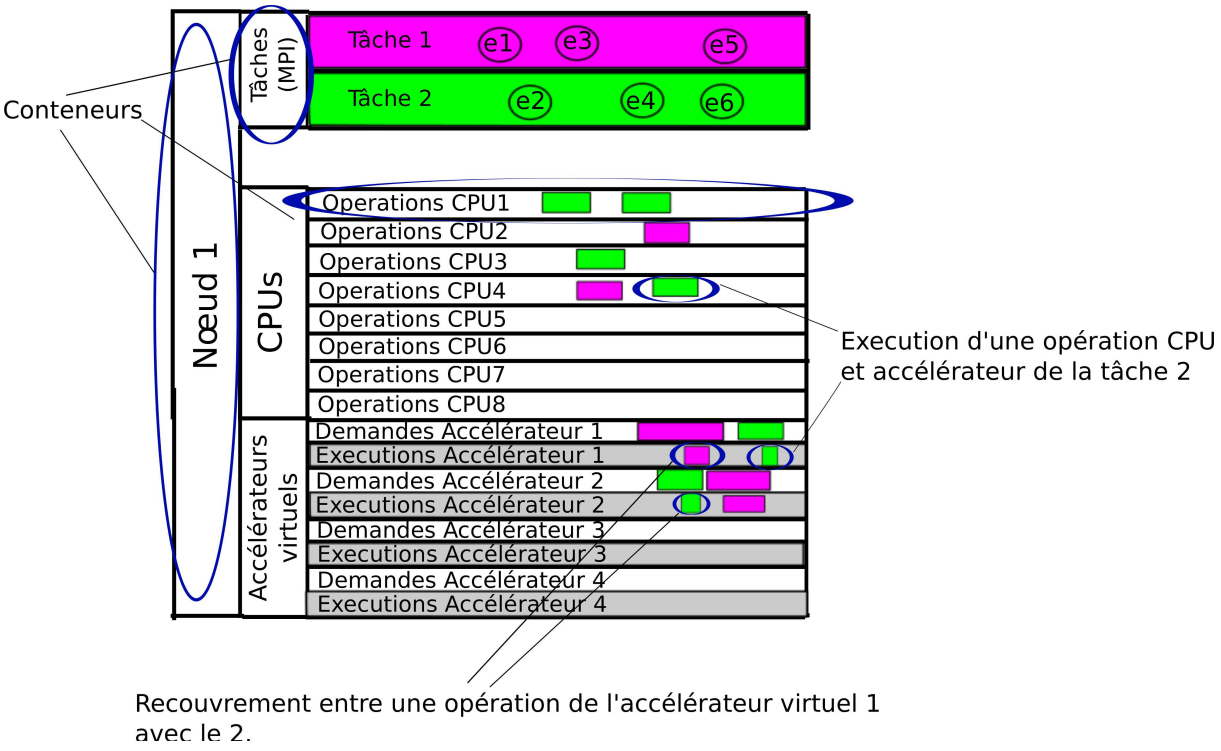


Figure 4.17 – Organisation de la visualisation à base de conteneurs

autant de conteneur de type “accélérateur” que d’accélérateurs virtuels et autant de conteneur de type “CPU” que de cœurs CPU.

— **Les opérations**, contenues dans le conteneur “type d’unité de calcul”, permettent de suivre l’activité de chaque unité de calcul. Une activité peut être soit un calcul soit un transfert mémoire et est représentée dans le conteneur “opérations” par un rectangle, proportionnel au temps de cette activité. Le conteneur “opération” des accélérateurs est scindé en deux (cf figure 4.17). Le premier contient les “demandes” et le second l’exécution de ces demandes. En effet, lorsque l’on se place dans un environnement virtualisé, il peut y avoir un certain temps entre la demande d’une opération accélérateur et son exécution proprement dite. Ce temps est dû au fait que plusieurs accélérateurs se partagent un même accélérateur physique. Cette particularité peut donc entraîner des délais qui sont mis en évidence ici.

— **Les tâches**, représentent les différentes tâches MPI ayant soumis du travail. Pour chaque tâche, il est possible de visualiser à quel moment sont soumis des calculs, et donc à quel moment la fonction `soumettre_threads` est appelée. Cela est représenté par un événement (marqué eX sur la figure 4.17). Grâce à cette information, il est possible de déterminer le temps entre la soumission et l’exécution proprement dite d’une requête. Chaque tâche possède une couleur. Nous avons choisi de faire correspondre la couleur d’une tâche à la couleur des activités du conteneur “opération”. Il est ainsi possible de déterminer depuis quelles tâches une activité a été lancée et donc sur quelles ressources de calcul sont déployées les différentes tâches.

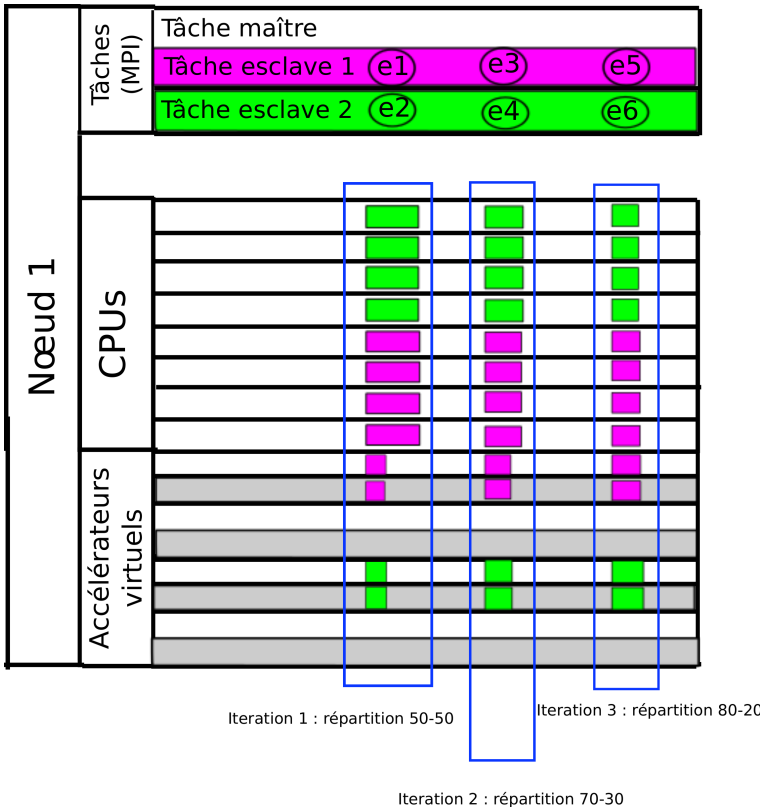


Figure 4.18 – Visualisation des trois itérations du produit scalaire

Visualisation d'opérations concurrentes La visualisation des opérations concurrentes sur les accélérateurs est réalisée grâce aux accélérateurs virtuels. Plusieurs accélérateurs virtuels pouvant se partager un même accélérateur physique, il est possible de voir un recouvrement calcul transfert en observant si au même moment un accélérateur virtuel calcule et un autre transfert des données. Sur la figure 4.17 nous considérons que les deux accélérateurs virtuels partagent le même accélérateur physique, et nous observons donc des opérations simultanées sur le même accélérateur.

4.3.4 Visualisation du calcul de produit scalaire

Nous reprenons ici notre exemple du produit scalaire en MPI. Comme précédemment, nous considérons que ce produit scalaire est un noyau de calcul appelé plusieurs fois et que le calcul hybride est équilibré à la troisième itération avec une répartition 80 - 20. La figure 4.18 montre ce que l'on veut observer pour les trois itérations du produit scalaire ; les répartitions indiquées sont donc celles montrées précédemment. Le programme est constitué de trois tâches MPI. La tâche maître contrôle l'exécution et ne lance aucun thread. Les deux tâches esclaves soumettent des requêtes pour exécuter un calcul de produit scalaire hybride, ce qui entraîne un déploiement de threads sur les accélérateurs et les CPU.

Nous donc avons trois tâches MPI, la tâche maître ne soumet rien et les deux tâches esclaves soumettent toutes les deux un déploiement de threads à chaque itération avec la

fonction `soumettre_threads`. Ces soumissions sont notées $e1$ - $e2$ pour l'itération 1, $e3$ - $e4$ pour l'itération 2 et $e5$ - $e6$ pour la dernière itération.

La performance d'une application est directement liée à la bonne utilisation des ressources de calcul. Dans le cas d'une architecture hybride l'utilisation conjointe de ressources CPU et accélérateur est un élément déterminant. De plus, les transferts entre la mémoire CPU et celle des accélérateurs sont coûteux et doivent être analysés avec soin.

Nous avons proposé un dispositif de visualisation qui permet au développeur d'évaluer la qualité de son application et son adéquation avec l'architecture. Nous nous sommes concentrés sur la partie hybride. Pour visualiser d'autres types d'informations, comme la qualité des communications MPI, il existe d'autres outils très puissants, par exemple, Intel Trace Analyser and Collector ou Vampir ; nous n'avons donc pas traité cet aspect.

Cependant, même si l'on ne s'intéresse qu'aux CPU et accélérateurs, la somme d'informations à visualiser peut être très importante. Ainsi, en fonction des abstractions utilisées, il est intéressant de réduire les informations affichées par notre outil de visualisation. Par exemple, si nous utilisons uniquement la virtualisation d'accélérateurs, il n'est pas nécessaire de montrer le travail des CPU. Nous verrons dans les chapitres suivants quelles sont les adaptations réalisées sur la partie visualisation en fonction des cas d'utilisation.

4.4 Conclusion

En rajoutant des accélérateurs aux cœurs CPU traditionnels, les machines hybrides complexifient considérablement le développement d'applications qui veulent tirer partie de toute la puissance disponible des grappes hybrides. Il est donc nécessaire d'adapter les applications pour qu'elle exploitent ce type d'architectures. L'objectif principal de cette thèse était de permettre à des simulations scientifiques existantes d'exploiter efficacement des architectures hybrides.

Ainsi, nous avons proposé d'étendre les modèles de programmation utilisés par une grande partie des codes de simulation existants pour y rajouter la possibilité d'utiliser des accélérateurs. Les modèles choisis sont ceux basés sur des paradigmes à passage de messages comme MPI et des modèles basés sur des threads comme OpenMP. Nous avons commencé par MPI, en proposant une abstraction destinée à virtualiser les accélérateurs pour permettre aux applications de voir une configuration standard. Nous avons ensuite étendu le modèle MPI + threads en ajoutant une notion de thread accélérateur et en proposant des mécanismes permettant de mettre en place des calculs s'exécutant simultanément sur plusieurs espaces mémoire disjoints. Pour analyser l'exécution des programmes utilisant nos modèles, nous avons également proposé des moyens d'observation hybrides. En étendant des modèles existants, nous pensons rendre l'intrusivité des modèles limitée dans les simulations existantes et ainsi permettre aux applications scientifiques d'exploiter les accélérateurs sans restructuration de leurs architectures. Nous verrons dans les deux derniers chapitres de cette thèse comment des applications scientifiques existantes peuvent utiliser nos modèles.

Nous avons abordé le problème en proposant une bibliothèque de fonctions (API) mise à disposition des développeurs. L'approche par API a l'avantage d'être assez standard : il suffit de construire une bibliothèque logicielle et la lier à une application. Elle peut s'avérer

néanmoins relativement lourde : il faut ajouter des appels de fonction et encapsuler du code dans d'autres fonctions. Des approches par directives de compilations existent et il pourrait être possible d'intégrer nos propositions dans des standards du type OpenMP ou OpenACC pour permettre aux utilisateurs d'implanter des calculs hybrides tels que nous les avons définis dans ce chapitre. On pourrait imaginer, par exemple, que le programmeur fournisse les fonctions de coupe et de groupement mais que les threads CPU et accélérateurs soient directement créés par le compilateur à partir d'un unique noyau de calcul.

Dans le chapitre suivant, nous décrivons deux bibliothèques logicielles, S_GPU 1 et S_GPU 2 qui permettent d'utiliser nos modèles dans des applications.

Chapitre Cinq

Mise en œuvre des propositions

Sommaire

Dans le chapitre précédent, nous avons défini un modèle de programmation hybride qui consiste à virtualiser des accélérateurs. Nous avons ensuite étendu ce modèle pour exploiter à la fois des CPU et des accélérateurs. Nous allons suivre dans ce chapitre la même démarche, et montrer comment nous avons mis en œuvre ces propositions dans deux bibliothèques ; la première, S_GPU 1 a pour but de donner l'illusion d'avoir autant d'accélérateurs que de tâches MPI. La deuxième, S_GPU 2 ajoute la prise en compte des CPU et permet l'exécution conjointe CPU – accélérateur.

Nous préciserons dans ce chapitre les interfaces de programmation et nous verrons le fonctionnement de ces deux bibliothèques.

5.1 Contexte de développement

Nous commencerons par décrire les environnements logiciels qui ont été employés pour construire ces bibliothèques. Nous verrons ensuite la stratégie que nous avons utilisée pour permettre aux applications Fortran d'utiliser nos bibliothèques. Nous finirons sur les techniques de traçage choisies.

5.1.1 Environnements utilisés

Les deux bibliothèques ont été programmées principalement avec l'aide du langage C++ avec quelques parties en C. Elles font appel à des fonctions système issues de la norme POSIX. Elles sont destinées à être utilisées sur des systèmes basés sur le noyau Linux, qui

est le système employé sur la majorité des grappes de calcul les plus performantes. En effet, un noyau Linux est employé à 92,40 % sur les 500 machines les plus puissantes d'après le classement TOP500 de juin 2012.

5.1.2 Interfaces de programmation

S_GPU 1 et S_GPU 2 étant codés en C et C++, elles peuvent naturellement être utilisées dans ces langages. Cependant, une grande partie des applications scientifiques est développée en Fortran : il est ainsi nécessaire de fournir au programmeur une interface de programmation accessible à partir de ce langage.

Problématique des arguments variables Au chapitre précédent, nous avons vu que le développeur définit des fonctions contenant le code d'un calcul. Ces fonctions sont ensuite soumises au support exécutif et exécutées par celui-ci. Il faut cependant noter que ces fonctions peuvent avoir un nombre quelconque d'arguments (par exemple deux tableaux d'entrée, un de sortie et une dizaine d'autres arguments contrôlant le calcul). Nos deux bibliothèques, qui mettent chacune en œuvre un support exécutif, doivent être capables de prendre en charge ce type de fonction à arguments variables. Considérons un code exécutable A associé à une liste de paramètres lp_A et un code exécutable B associé à la liste lp_B. Nos bibliothèques, doivent être capables d'exécuter indifféremment $A(lp_A)$ et $B(lp_B)$ de manière générique.

Mise en place dans S_GPU 1 et S_GPU 2 Il existe dans le langage et la bibliothèque standard du C une possibilité d'utiliser des fonctions avec un nombre variable d'arguments. Néanmoins, ces fonctions ne sont pas appelables directement depuis un code Fortran. Ainsi, pour nos bibliothèques, nous avons choisi de sérialiser les différents arguments nécessaire aux fonctions dans une unique structure C.

Pour mettre en place cette sérialisation, nous proposons au programmeur de soumettre des fonctions prenant un unique argument générique de type `void *arg` ; l'argument arg contient la structure C sérialisant les différents paramètres nécessaires à la fonction. C'est une stratégie semblable à celle définie par POSIX pour passer des paramètres lors de la création d'un thread avec la fonction `pthread_create`.

Exemple d'un code accessible depuis Fortran Nous allons maintenant montrer les étapes qu'il faut réaliser pour qu'un code Fortran puisse soumettre des fonctions à nos bibliothèques.

Supposons qu'une fonction de calcul `calc(T1 arg1, T2 arg2, T3 arg3)` avec trois arguments de type T1, T2 et T3, doive être exécutée par une de nos bibliothèques depuis un code Fortran. L'idée de base est d'appeler un code C depuis Fortran et de déléguer à ce code C la création des paramètres. Les deux extraits de programme 5.2 (en C) et 5.1 montrent les trois actions à réaliser pour permettre à nos supports exécutifs d'exécuter la fonction `calc` :

1. **Adapter le code Fortran** Le code Fortran doit être adapté pour pouvoir appeler la fonction `bind_calc` décrite à l'étape 2. Il s'agit d'un appel de fonction, représenté sur le programme 5.1.

Listing 5.1 – Le code fortran

```
1 call bind_calc(arg1,arg2,arg3,ret_val)
```

2. **Créer une fonction appelable depuis Fortran** Une nouvelle fonction, nommée `bind_calc` (ligne 16), destinée à être appelée depuis le code Fortran de l'étape 1 doit être écrite. Elle a pour but de créer un objet de type `struct arg_calc` (ligne 18) qui est une structure C serialisant les arguments nécessaires à `calc`. Un appel à la fonction `soumettre_calcul` (ligne 20) permet de soumettre la fonction. Les deux arguments sont la fonction `exec_calc` et les arguments encapsulés dans la structure de type `struct arg_calc`. Lorsque `exec_calc` sera exécutée, la fonction `calc` aura accès aux bons arguments.
3. **Créer une fonction prenant en compte les arguments variables** La fonction `calc` doit être encapsulée dans une fonction qui prenne en arguments un `void *arg` représentant la structure contenant les arguments sérialisés de `calc`. La ligne 10 du programme 5.2 permet d'interpréter le paramètre `void *arg` comme une structure et lance la fonction `calc` avec les bons arguments (ligne 12).

S_GPU 1 et S_GPU 2 fournissent des fonctions semblables à `soumettre_calcul` lorsqu'une fonction écrite par le programmeur doit être soumise. Cette approche a l'avantage de fonctionner avec la plupart des compilateurs Fortran et C disponibles sur les grappes hybrides (nous avons validé avec GNU gfortran et Intel ifort). Il faut bien noter que ce passage entre C et Fortran est uniquement nécessaire pour l'exécution de fonctions fournies par l'utilisateur. Toutes les autres fonctions de l'API (transferts, allocation mémoire, initialisation...) sont appelables directement depuis un code Fortran.

5.1.3 Instrumentation et traçage

Chacune des fonctions des bibliothèques S_GPU 1 et 2 a été prévue pour enregistrer des traces afin d'être capable de suivre le déroulement des applications les utilisant. Nous avons employé la bibliothèque GTG [1] pour générer les traces. Pour chaque évènement, comme un calcul ou un transfert, une information est écrite dans un fichier de trace. Dans le cas d'une grappe hybride, un fichier de trace est enregistré sur chaque nœud.

Chaque nœud a une horloge indépendante, et, en l'absence de synchronisation nous pouvons observer des décalages entre les traces des différents nœuds. Cependant, nous nous intéressons à la visualisation de problèmes de performance, nous n'avons donc pas besoin d'avoir une grande précision concernant la date de début ou de fin d'un évènement particulier ; nous avons uniquement besoin de savoir si les ressources ont été correctement utilisées. Ainsi, comme les horloges des différents nœuds d'une grappe sont régulièrement synchronisées entre elles, cette précision est satisfaisante et une concaténation des différents fichiers suffit pour produire une unique trace représentant l'ensemble de la grappe hybride.

La visualisation de ces traces est effectuée grâce au logiciel Vite [2] qui est capable d'interpréter les traces générées par GTG et de les afficher.

1. `gtg.gforge.inria.fr/`
2. `vite.gforge.inria.fr/`

Listing 5.2 – Interface en C

```
1  struct arg_calc
2  {
3  T1 arg1;
4  T2 arg2;
5  T3 arg3;
6  };
7
8  void exec_calc(void *arg)
9  {
10 struct arg_calc *arg = (struct arg_calc*)arg;
11
12 calc(arg->arg1,arg->arg2,arg->arg3);
13 }
14
15
16 void  bind_calc(T1 *arg1, T2 *arg2, T3 *arg3, T_{ret} *retour)
17 {
18   struct arg_calc arg {*arg1,*arg2,*arg3};
19
20    *retour = soumettre_calcul(&exec_calc,&arg);
21 }
```

La suite de ce chapitre a pour but de présenter les choix de mise en œuvre employés pour S_GPU 1 et S_GPU 2.

5.2 S_GPU 1 : Virtualisation et partage de GPU

Nous commençons par décrire S_GPU 1, qui consiste donc en une bibliothèque logicielle mettant en œuvre notre modèle de programmation basé sur la virtualisation des accélérateurs. Il s'agit donc de donner l'illusion à chaque tâche MPI d'un nœud qu'elle possède son propre accélérateur. Dans la mise en œuvre de S_GPU 1 nous avons traité uniquement les accélérateurs de type GPU programmables en CUDA.

5.2.1 Virtualisation des accélérateurs

S_GPU 1 est une couche logicielle qui se place entre une application et le GPU (fig 5.1). Chaque processus (tâche MPI par exemple) exécute ses opérations sur le GPU en utilisant l'interface de programmation fournie par S_GPU 1.

Il y deux étapes principales pour mettre en place la virtualisation des GPU décrite au chapitre précédent. La première consiste à attribuer aux tâches MPI un GPU virtuel, et la seconde consiste à partager les accélérateurs entres les tâches MPI.

Attribution des GPU aux tâches MPI Lors de l'appel de `sg_init()`, qui est la fonction d'initialisation de la bibliothèque S_GPU 1, chaque processus MPI va se voir attribuer un unique GPU. L'attribution se déroule en deux phases. Dans la pre-

mière phase, chaque processus impliqué dans le calcul s'enregistre auprès de la bibliothèque. Une fois les processus connus, les GPU sont attribués de manière équitable aux processus. Par exemple, s'il y a huit processus et deux GPU, quatre processus auront le premier GPU, les quatre autres le second. L'attribution est ici automatique et il n'y a pas d'intervention du programmeur : c'est le comportement par défaut.

Il est possible de mettre en place une attribution manuelle. Pour cela, un fichier de configuration, appelé `GPU.conf` permet de spécifier des informations d'affinité et de demander qu'un processus utilise un GPU particulier. Ce fichier autorise également de limiter le nombre d'accélérateurs à utiliser pour un ensemble de tâches MPI. Ainsi, si nous sommes dans une configuration où il y a quatre GPU et toujours huit tâches MPI, S_GPU 1 peut être configuré pour utiliser uniquement deux GPU : 4 tâches MPI se partageront le GPU 1 et les 4 autres le GPU 2. Les GPU 3 et 4 seront inutilisés par S_GPU 1.

Partage des accélérateurs Après la phase précédente, chaque processus connaît son GPU virtuel. Les GPU virtuels doivent donc partager le GPU physique de la machine.

Pour partager les GPU, nous avons mis en place l'approche FIFO présentée au chapitre précédent. Nous avons pour cela décidé d'utiliser des sémaphores UNIX, qui fonctionnent en mode FIFO. Étant donné qu'il est possible d'avoir des transferts et des calculs simultanément sur les GPU NVIDIA, les processus qui se partagent un GPU utilisent deux sémaphores ; le premier pour gérer les transferts concurrents en mémoire, le second pour gérer les demandes de calcul concurrentes. Pour éviter un partage des bus PCI-express, qui peut être inefficace, nous interdisons à deux processus, connectés au même GPU, de faire simultanément des transferts mémoire. Nous avons ainsi un contrôle complet sur le partage de la bande passante des bus PCI-express. Néanmoins, si un processus a le sémaphore « calcul », un autre processus peut faire un transfert simultanément si le sémaphore « transfert » est libre.

Les GPU de type TESLA ne supportent pas, au niveau matériel, l'exécution concurrente de noyaux de calcul. Ainsi, si plusieurs noyaux de calcul sont soumis simultanément à un GPU, l'environnement CUDA va ordonnancer ces noyaux en sérialisant leurs exécutions sur le GPU. Nous n'avons cependant aucun contrôle sur l'ordonnancement réalisé par l'environnement CUDA. Nous avons donc décidé de gérer nous-mêmes cet ordonnancement, à savoir, ne pas laisser plusieurs processus lancer un calcul simultanément sur le même GPU. Comme CUDA, nous sérialisons l'exécution des noyaux de calcul sur le GPU. Cependant, comme nous contrôlons l'ordonnancement, nous savons exactement à quel moment un kernel est exécuté, ce qui nous permet d'avoir des informations précises pour la génération des traces d'exécution.

S_GPU 1 a été développé pour les GPU de la génération TESLA. Les processeurs graphiques des générations FERMI et KEPLER autorisent une exécution concurrente des noyaux de calcul sur GPU et ont pour cela un ordonnanceur relativement évolué, comme présenté au chapitre 3. Sur ce type de matériel, conserver un or-

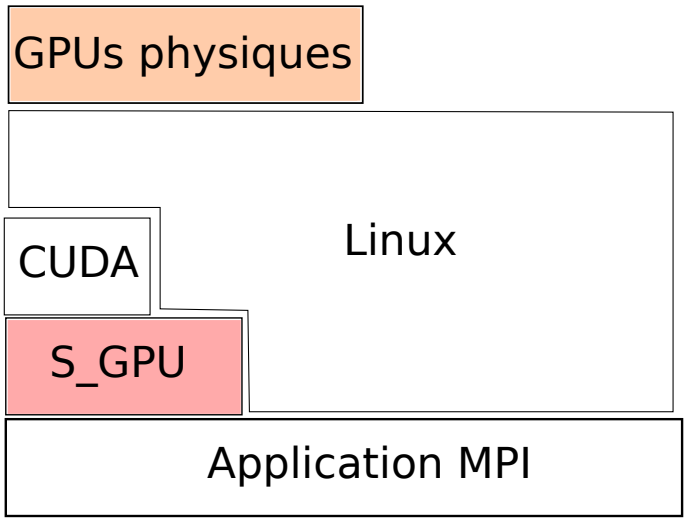


Figure 5.1 – S_GPU 1 se place entre l'application et les processeurs graphiques (GPU)

donnancement au niveau d'une bibliothèque logicielle a un sens. En effet, notre structuration en threads GPU nous permet d'avoir des informations, comme les dépendances entre threads, qui seraient inaccessibles à un ordonnanceur bas niveau comme celui des GPU. Ces dépendances nous permettraient, par exemple, de déterminer des opérations indépendantes pouvant être exécutées simultanément sur un GPU. Ainsi, en adaptant S_GPU 1 pour autoriser l'exécution simultanée de plusieurs noyaux de calcul sur un même GPU, il serait tout à fait envisageable de supporter efficacement les nouvelles générations d'accélérateurs.

5.2.2 Programmation avec S_GPU 1

S_GPU 1 se place au même niveau d'abstraction que CUDA (ou OpenCL). Les exemples de code présentés dans cette partie sont tous en Fortran et proviennent d'une version de BigDFT qui utilise S_GPU 1 (l'utilisation de S_GPU 1 avec BigDFT sera vue en détail au chapitre 7).

Threads GPU Pour soumettre des requêtes dans S_GPU 1, nous avons utilisé la notion de threads GPU décrite en 4.2.3.1. Ainsi, plutôt que de soumettre des transferts mémoire et des calculs de manière non-structurée, nous avons choisi d'organiser les soumissions sous la forme de threads GPU. Les threads GPU contiennent une succession d'opérations GPU et des opérations de manipulation de données. Les threads créés sont soumis à S_GPU 1.

Pour exécuter les threads soumis, il suffit d'appeler une fonction de la bibliothèque : `launch_all_threads()`. La fonction bloque l'exécution du programme tant que toutes les opérations des threads ne sont pas terminées.

Interface de programmation

`sg_init()` Initialise les processus d'un même nœud et les associe aux processeurs graphiques.

```
1       call sg_create_thread(thread_ptr)
2
3       do iorb=1,num_orbs
4          call sg_mem_copy(...,
5                           thread_ptr)
6          call sg_gpu_send(...,
7                           thread_ptr)
8       end do
9       call calcul(...,
10                  thread_ptr)
11      call sg_gpu_recv(...,
12                           thread_ptr)
13      call sg_mem_copy(...,
14                           thread_ptr)
15
16      call sg_launch_all_threads()
```

`sg_create_thread(threadPtr)` Crée un nouveau thread et renvoie un pointeur sur celui-ci.

`sg_launch_all_threads()` Exécute toutes les instructions placées dans les threads créés. Cette opération est bloquante. Elle finit lorsque tous les threads ont été exécutés.

`sg_cpu_pinned_allocation(int size, CPUptr, errCode)` Alloue de la mémoire punaisée sur la mémoire centrale du CPU. Nous verrons plus loin que cette fonction est indispensable pour les recouvrements entre les calculs et les transferts sur GPU.

`sg_gpu_send(int nsize, CPU_pointer_pi, GPU_pointer, errCode, threadPtr)` Ajoute dans le thread passé en paramètre une instruction permettant de faire un transfert mémoire du CPU au GPU. Il existe une fonction symétrique pour le transfert des données du GPU vers le CPU.

`calcul(param...)` Exécute un calcul sur le GPU. Le passage de paramètres utilise la stratégie présentée en 5.1.2.

Exemple de code Cet exemple, issu de l'application BigDFT, est certes simplifié mais conserve le même squelette que le code réel. Il crée un thread GPU, effectue plusieurs transferts mémoire du CPU au GPU, lance un calcul et rapatrie les données du GPU vers la mémoire CPU. Il faut noter qu'il y a deux appels à `sg_mem_copy` (ligne 4 et 13). Cette fonction effectue une copie d'un tableau de données vers une zone mémoire CPU allouée par `sg_cpu_pinned_allocation`; les transferts GPU → CPU (ligne 6) ou CPU → GPU (ligne 11) se font à partir de cette zone mémoire « punaisée », c'est-à-dire des pages mémoire bloquées que le système de pagination ne peut pas réutiliser. Ce type de mémoire est nécessaire avec CUDA pour pouvoir recouvrir les transferts et les calculs — i.e. faire des transferts mémoire en même temps que des calculs —.

La ligne 16 (bloquante) permet l'exécution de toutes les commandes empilées dans le flot créé.

5.2.3 Résumé

S_GPU 1 est notre première proposition de support exécutif destiné à virtualiser des ressources GPU. Il permet aux simulations scientifiques d'utiliser une configuration plus homogène. Cette bibliothèque ne prend pas en compte l'exécution conjointe d'un même noyau de calcul sur les CPU et les GPU. Nous avons présenté, dans le chapitre précédent, un modèle de programmation basé sur l'exécution hybride de noyaux de calcul ; la bibliothèque présentée dans la suite de chapitre met en place ce modèle.

5.3 S_GPU 2 : Exécution hybride de noyaux de calculs

Cette section est consacrée à S_GPU 2, qui est la bibliothèque logicielle qui met en œuvre notre modèle de programmation hybride. S_GPU 2 permet donc de mettre en place un calcul hybride, c'est-à-dire un calcul qui utilise simultanément des ressources CPU et GPU. S_GPU 2 propose au programmeur de soumettre des threads CPU et des threads GPU afin de lancer un calcul hybride. Ce modèle de programmation hybride a nécessité la conception d'un nouvel environnement logiciel que nous avons appelé S_GPU 2. S_GPU 2 fournit une interface de programmation qui permet d'employer notre modèle de programmation (soumission de threads, synchronisation). Cette interface est le moyen d'accéder à l'environnement d'exécution.

Nous décrivons, tout d'abord, l'environnement d'exécution de S_GPU 2, suivent des compléments sur les interfaces de programmation de la bibliothèque.

5.3.1 Environnement d'exécution hybride CPU – accélérateur

Nous avons entrepris la mise en place d'un environnement d'exécution qui consiste à réaliser des opérations d'allocation de ressources, de régulation de charges et d'exécution de threads CPU et GPU.

Nous allons, dans un premier temps, décrire l'organisation générale de l'implémentation de cet environnement d'exécution dans S_GPU 2. Ensuite, nous allons détailler comment le code des threads, fourni par l'utilisateur, est exécuté. Nous finirons sur la gestion de la mémoire accessible aux threads.

5.3.1.1 Organisation générale

L'environnement d'exécution met en place l'algorithme 1 de la section 4.2.3.3, et se base donc sur trois queues, chacune contenant les soumissions CPU, GPU et hybrides. Comme spécifié dans l'algorithme, S_GPU 2 scrute successivement ces trois queues et alloue les ressources matérielles indiquées dans les soumissions.

Deux choix sont envisageables pour mettre en œuvre l'environnement d'exécution de S_GPU 2.

— Le premier consiste en une bibliothèque logicielle dont l'environnement d'exécution est distribué dans l'ensemble des processus MPI utilisant S_GPU 2. Le contrôle est distribué et les processus communiquent entre eux pour assurer la cohérence globale de l'ensemble.

— L'autre approche est centralisée : chaque processus MPI s'adresse à un processus central qui prend toutes les décisions de manière centralisée. Le contrôle est donc ici centralisé. Ce modèle est efficace avec un nombre limité de tâches MPI.

Notre modèle est basé sur une programmation hybride composée de tâches MPI associés à des threads OpenMP. Avec ce mode de programmation, nous suggérons de créer un nombre de tâches lié à la topologie de la plate forme considérée. Par exemple sur un nœud comportant 4 processeurs de 6 cœurs chacun, associés à 4 bancs NUMA, 4 tâches MPI peuvent être lancées. Chacune créant 6 threads. En utilisant ce type d'approche, il est envisageable d'avoir peu de tâches MPI, même sur des nœuds de plusieurs centaines de cœurs.

Le nombre de tâches MPI étant limité sur un nœud, nous avons choisi le modèle centralisé dans S_GPU 2. Ainsi, S_GPU 2 est composé d'une bibliothèque logicielle fournissant des interfaces de programmation (API [3]) ainsi que d'un processus S_GPU 2, représenté sur la figure 5.2. Il y a donc un processus S_GPU 2 par nœud d'une grappe hybride. Les fonctions d'accès aux trois queues ont été conçues pour être *thread-safe* afin que plusieurs tâches MPI concurrentes puissent soumettre simultanément des requêtes.

Les tâches MPI soumettent ainsi des threads (CPU ou GPU) contenant un code exécutable et le processus S_GPU 2 alloue des ressources de calcul aux threads. Une fois ces ressources allouées, le code des threads est exécuté.

5.3.1.2 Exécution des threads S_GPU 2

Le processus S_GPU 2 met en place l'allocation des ressources et se base donc sur l'algorithme 1 de la section 4.2.3.3. Il décide ainsi quand un thread, qui a été soumis par le programmeur via une tâche MPI, doit être lancé. Nous décrivons dans cette section les choix que nous avons effectués pour mettre en place l'exécution des threads dans S_GPU 2. Nous verrons tout d'abord comment le code des threads est déclenché. Les tâches MPI et le processus S_GPU 2 doivent communiquer : nous détaillerons donc dans un second temps les différents signaux échangés entre les tâches MPI et le processus S_GPU 2. Enfin nous donnerons des précisions concernant l'ordonnancement des threads S_GPU 2.

Déclenchement du code des threads Lorsque l'on utilise, S_GPU 2, les tâches MPI et le processus S_GPU 2 sont lancés sur le nœud. Nous allons montrer ici qui déclenche le code des threads dans S_GPU 2.

— Il est tout à fait possible de faire déclencher le code des threads directement par le processus MPI soumettant la requête : les processus signalent à S_GPU 2 qu'ils désirent exécuter du code, et attendent une réponse de S_GPU 2. Le code des threads est exécuté directement dans l'espace mémoire de la tâche MPI ce qui facilite les communications entre la tâche MPI et les threads. Néanmoins, les tâches MPI et le processus S_GPU 2 doivent communiquer pour déterminer quand lancer un thread et quand ce thread est fini.

— Une autre approche, opposée à la précédente, consiste à faire déclencher le code par le processus S_GPU 2 : les tâches MPI soumettent des requêtes contenant du

3. Application Programming Interface

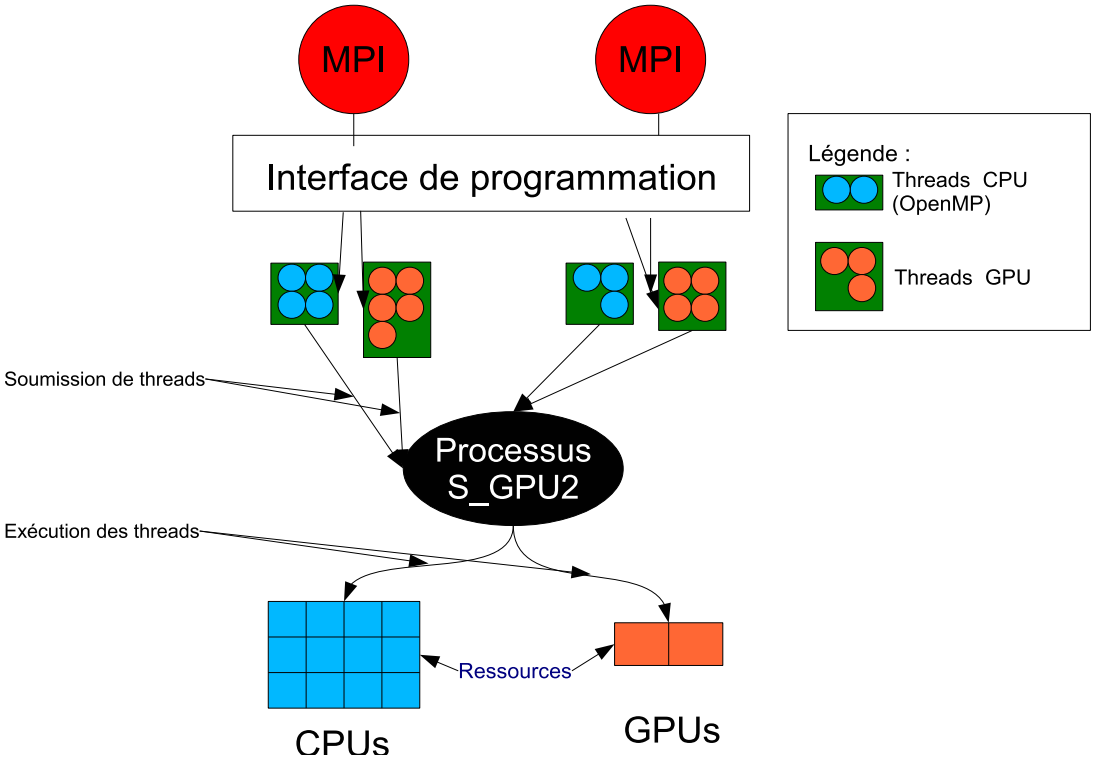


Figure 5.2 – Architecture générale de S_GPU 2

code exécutable et attendent le déclenchement du code soumis. Le code n'est donc pas déclenché à partir de l'espace des processus de l'application mais à l'intérieur du processus S_GPU 2. Cette approches nécessite d'utiliser des mécanismes de mémoire partagée puisque les threads ne sont plus exécutés dans l'espace mémoire du processus soumettant le thread.

L'avantage de tout déclencher sur les processus soumettant les requêtes est d'avoir un processus S_GPU 2 relativement simple. Cependant, le problème avec cette approche est de limiter certaines optimisations possibles sur le matériel. Par exemple, pour exécuter plusieurs calculs simultanément sur un GPU FERMI, ces calculs doivent être lancés depuis le même processus.

Nous avons choisi dans S_GPU 2 de déclencher le code des threads GPU par le processus S_GPU 2. Le code des threads CPU, quant à lui, n'est pas déclenché par le processus S_GPU 2 mais directement depuis l'application.

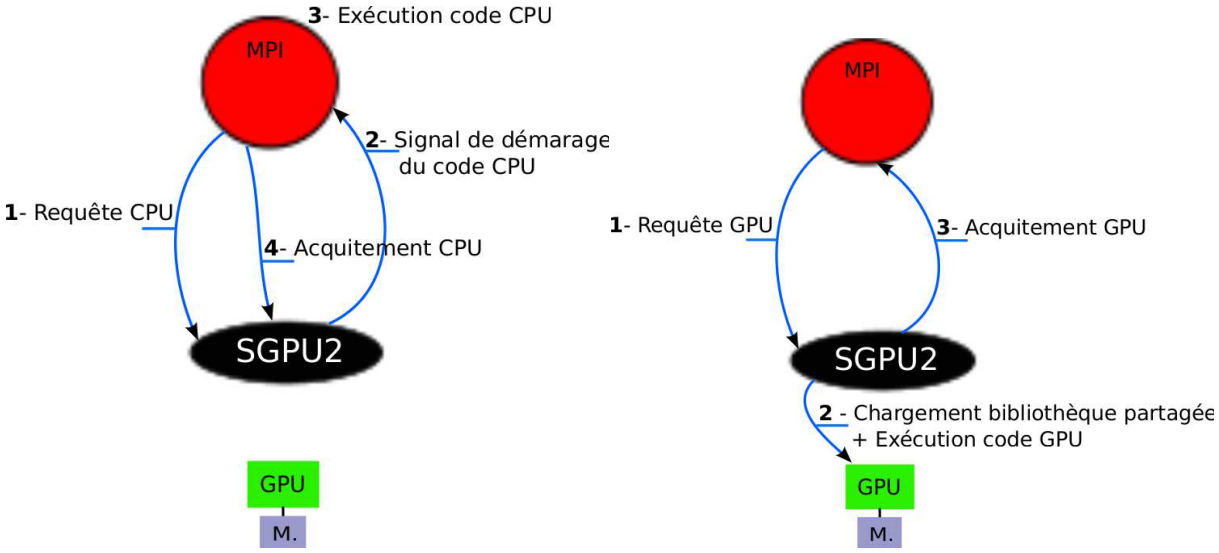
Utilisation de bibliothèques partagées Le déclenchement du code GPU étant effectué dans l'espace mémoire du processus S_GPU 2, il est nécessaire que celui-ci ait un moyen d'exécuter un code fourni par une tâche MPI de l'application.

La première option serait de compiler directement le code dans le processus S_GPU 2. Ainsi, à chaque application, un processus S_GPU 2 spécifique doit être généré. La solution devient non-générique (il y a un processus par application), et si deux applications souhaitent utiliser simultanément S_GPU 2, cela devient très difficile car il faudra générer un processus S_GPU 2 contenant le code attendu pour les deux applications.

L'approche choisie pour S_GPU 2 est l'utilisation de bibliothèques partagées dynamiques. Ce type d'objet est un fichier contenant du code exécutable pouvant être chargé dynamiquement par un programme et exécuté par celui-ci. Ces fichiers ont conventionnellement l'extension `.so` sur un système Linux. L'idée est donc de mettre le code exécutable dans une bibliothèque partagée et de préciser lors de chaque requête dans quelle bibliothèque partagée se trouve le code à exécuter. Lorsque S_GPU 2 décide de déclencher un code, il suffit qu'il charge la bibliothèque partagée correspondante et exécute le code contenu dans celle-ci. L'avantage principal de cette technique est l'indépendance totale du processus S_GPU 2 vis à vis des applications. Il faut en contrepartie compiler du code dans une bibliothèque partagée.

Interactions entre les processus Afin de contrôler l'exécution du code sur les GPU ou les CPU, plusieurs interactions ont lieu entre le processus soumettant des requêtes et le processus S_GPU 2. Les interactions sont basées sur des outils de synchronisation standard, à savoir, des variables de condition ainsi que des sémaphores. La figure 5.3 montre les deux types d'exécution implantés dans S_GPU 2. L'exécution du code CPU est réalisée par le processus soumettant des requêtes (figure 5.3a), alors que le code GPU est exécuté directement par le processus S_GPU 2 via une bibliothèque partagée (figure 5.3b).

– **Exécution CPU** Lors de l'exécution d'un code multithreadé CPU, représenté sur la figure 5.3a, la première étape est la soumission d'une requête de création de threads CPU par le programmeur (étape 1). Lorsque le thread peut être exécuté, S_GPU 2 commande à l'application d'exécuter le code CPU du thread (étape 2). Le message envoyé contient également la liste des processeurs sur lesquels le code peut



(a) Exécution d'un code CPU **(b)** Exécution d'un code GPU

Figure 5.3 – Exécution du code dans S_GPU 2

s'exécuter. Une fois le code exécuté (étape 3), l'application le signale à S_GPU 2 (quatrième et dernière étape).

– **Exécution GPU** L'exécution d'un code GPU (figure 5.3b) commence également par la soumission d'une requête de création de threads GPU (étape 1). Cette requête contient des informations concernant la bibliothèque partagée dans laquelle se trouve le code du thread à exécuter. Une fois que S_GPU 2 décide d'exécuter le code, l'étape 2 est lancée : elle consiste en un chargement de la bibliothèque suivie de l'exécution du code. Une fois l'exécution du code terminée, l'application en est avertie (étape 3).

– **Exécution Hybride** Pour une exécution hybride, les threads CPU sont lancés depuis l'application et les threads GPU depuis le processus S_GPU 2. Nous considérons que les fonctions de coupe et de regroupement sont des fonctions CPU, elles sont exécutées de la même manière que le code des threads CPU.

Ordonnancement des threads S_GPU 2 Nous avons vu jusqu'à présent que le code des threads était déclenché soit par la tâche MPI soit par S_GPU 2 et que des messages étaient échangés entres les différents processus. Nous allons maintenant décrire plus précisément comment les threads CPU et GPU sont ordonnancés.

– **Thread GPU (CUDA ou OpenCL)** L'ordonnancement des threads GPU suit la stratégie présentée en 4.2.3.3, qui est basée sur l'utilisation d'un coloriage. Les threads sont donc coloriés afin que plusieurs threads, désirant partager de la mémoire GPU, puissent utiliser le même GPU.

Certains threads peuvent contenir des transferts mémoire et d'autres des calculs ; il est donc indispensable d'autoriser un recouvrement transfert – calcul, entre plusieurs threads. De plus, les processeurs GPU de la génération FERMI permettent une exécution concurrente de noyaux de calcul que nous avons décidé d'exploiter. Bien sûr, en fonction de la génération des GPU utilisés, la concurrence d'accès aux ressources est variable : sur des FERMI, plusieurs calculs peuvent être exécutés simultanément sur un même processeur GPU, alors que dans les générations précédentes, seul un calcul et un transfert peuvent être en concurrence. Il est donc nécessaire d'utiliser un mécanisme générique permettant de lancer simultanément plusieurs opérations sur un GPU et qui exploite le niveau de concurrence supporté par le matériel.

Avec CUDA, une méthode très utilisée pour lancer plusieurs opérations simultanément se base sur l'utilisation de *streams* CUDA. Un stream CUDA peut être vu comme un flot dans lequel des commandes CUDA, comme des transferts mémoire ou des calculs, sont exécutés successivement. La soumission d'opérations dans un stream est asynchrone : il est donc possible de créer parallèlement plusieurs streams, chacun exécutant des opérations en concurrence sur le GPU. S_GPU 2 maintient ainsi plusieurs streams CUDA simultanément, chaque stream étant associé à un thread.

Dans le cas de threads exécutant du code CUDA, nous avons utilisé les streams CUDA afin d'exécuter parallèlement différents threads GPU. Un stream CUDA est passé à chaque invocation des kernels du thread GPU et plusieurs threads indépendants sont donc lancés simultanément sur le même GPU. Afin de déterminer la fin de l'exécution de ce type de thread, nous faisons de l'attente active en appelant régulièrement la fonction CUDA `cudaStreamQuery`, qui permet de connaître l'état d'un stream à tout moment.

Pour les threads exécutant du code OpenCL, nous utilisons une méthode similaire en utilisant des `command queue` [OPE11a] plutôt que des streams CUDA. Une `command queue` est générée par S_GPU 2 et passée à chaque fonction OpenCL du thread.

– **Thread CPU** L'ordonnancement des threads CPU est de la responsabilité du système d'exploitation ou d'une bibliothèque spécialisée, S_GPU 2 se contente donc de lancer le code CPU multithreadé sur les ressources allouées et attend la fin de son exécution. Au moment du déclenchement des threads sur la tâche MPI, un thread est créé dans le processus S_GPU 2 qui se bloque sur un sémaphore. Une fois que la tâche MPI finit l'exécution du code soumis, elle débloque le sémaphore et donc également le thread du côté S_GPU 2 qui attendait sur ce sémaphore. Ainsi, le thread peut initier la fonction de regroupement dans le cas d'une soumission hybride, et libérer les ressources de calcul allouées.

Nous avons vu que le code des threads est lancé soit depuis le processus S_GPU 2 pour les threads GPU, soit depuis la tâche MPI pour les threads CPU. Cependant, les threads doivent également pouvoir accéder à des données contenues en mémoire ; il reste

maintenant à montrer comment la mémoire est gérée avec S_GPU 2 pour qu'elle soit accessible aux threads.

5.3.1.3 Gestion de la mémoire par les threads

Les espaces mémoire entre CPU et GPU sont disjoints et des mouvements de données sont donc nécessaires pour faire transiter des données d'un espace à un autre. Dans S_GPU 2, l'exécution du code des threads GPU est initiée par le processus S_GPU 2 et non pas par la tâche MPI qui soumet une requête. Nous allons donc montrer comment les transferts sont réalisés sachant qu'il y a un intermédiaire – le processus S_GPU 2 – entre la tâche MPI et le GPU.

La figure 5.4 montre la différence entre une application classique, qui initie directement un transfert de la mémoire centrale vers le GPU, et une application S_GPU 2, où se trouve un intermédiaire. Avec notre approche, l'application soumet une requête de transfert et c'est le processus S_GPU 2 qui se charge de faire ce transfert.

Deux approches sont possibles pour réaliser ces transferts mémoire.

1. Le processus MPI envoie des données à S_GPU 2, et S_GPU 2 transfère ces données aux accélérateurs.
2. S_GPU 2 a accès à la zone mémoire à transférer dans l'espace mémoire du processus MPI et l'envoie directement aux accélérateurs.

L'approche 1 est relativement simple mais demande deux copies mémoire : une première depuis les processus MPI vers le processus S_GPU 2 du nœud et une deuxième depuis S_GPU 2 vers les accélérateurs. Cette approche nécessite également de dupliquer les zones mémoire allouées : la première sur la tâche MPI, la deuxième dans le processus S_GPU 2. L'approche 2 ne nécessite qu'une copie mémoire : du processus MPI vers l'accélérateur, le tout contrôlé par S_GPU 2. Elle ne nécessite pas non plus de dupliquer les zones mémoire allouées.

Afin de minimiser les copies mémoire, coûteuses, nous avons choisi l'approche 2 qui demande donc à l'application d'allouer ses données de telle sorte que le processus S_GPU 2 puisse y accéder. Le moyen que nous avons utilisé est l'emploi d'APIs POSIX permettant de partager de la mémoire entre différents processus. Il s'agit notamment des fonctions `shm_open` et `mmap`. L'API de S_GPU 2 fournit donc des fonctions permettant d'allouer, depuis une tâche MPI quelconque, une zone de mémoire partagée entre le processus MPI et le processus S_GPU 2. Comme nous l'avons vu, les transferts entre mémoire CPU et GPU sont les plus performants lorsque des zones de mémoire "punaisées", enregistrées dans le driver CUDA, sont utilisées. Depuis la version 4.0 de CUDA, une nouvelle fonctionnalité permet d'enregistrer et de rendre punaisée une zone mémoire déjà allouée. Ainsi, il est possible de rendre punaisée les zones de mémoire partagées allouées par S_GPU 2.

Ainsi, une application S_GPU 2 doit utiliser des fonctions de l'API de notre bibliothèque pour allouer des zones mémoire qu'elle souhaite envoyer aux GPU.

Dans cette section, nous avons montré des éléments d'architecture de S_GPU 2. Ces caractéristiques sont importantes car le programmeur doit comprendre le fonctionnement S_GPU 2 pour utiliser l'interface de programmation de manière avancée.

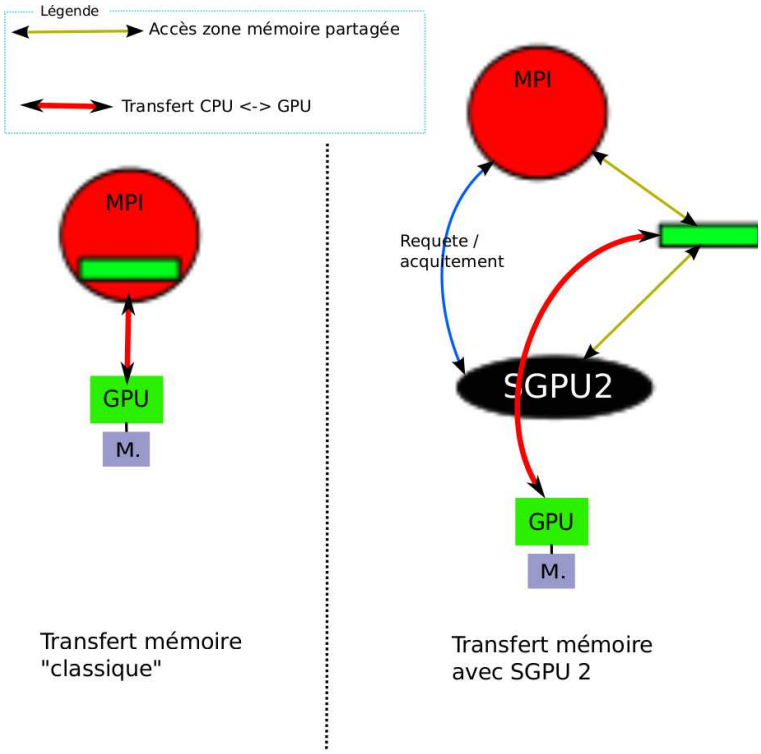


Figure 5.4 – Transferts mémoire pour une application classique et une application S_GPU 2

5.3.2 Utilisation avancée de l'interface de programmation

Les éléments d'architecture de S_GPU 2 que nous avons vus jusqu'à présent impliquent quelques particularités dans l'utilisation de l'interface de programmation. Nous n'allons pas ici exposer de nouveau l'interface de programmation qui a déjà été présentée au chapitre précédent. Mais nous allons montrer comment réaliser les allocations en mémoire partagée en C et en Fortran et détailler la création des threads GPU en montrant notamment l'utilisation des bibliothèques partagées.

5.3.2.1 Allocation des zones mémoire partagées

Pour allouer des zones mémoire partagées entre le processus S_GPU 2 et l'application en langage C, nous avons redéfini l'opérateur `malloc`. Il s'agit donc d'utiliser un allocateur fourni par S_GPU 2 plutôt que celui de la bibliothèque standard du C. Ce nouvel allocateur va renvoyer un pointeur vers une zone de mémoire partagée accessible à S_GPU 2. Il va également enregistrer la zone mémoire dans l'environnement CUDA et la rendre punaisée.

En Fortran, l'allocation de mémoire dynamique est traditionnellement réalisée grâce à l'opérateur `allocate`. Malheureusement, la norme Fortran ne permet pas la redéfinition de cet opérateur : il est donc impossible, en respectant cette norme, d'employer `allocate` pour allouer de la mémoire partagée utilisable avec S_GPU 2.

Nous allons proposer, dans la suite de cette section, un moyen pour allouer de la mémoire partagée accessible à S_GPU 2 depuis programme Fortran.

Listing 5.3 – Code Fortran original

```
1 program exemple
2 real(kind=4), dimension(:,:),allocatable :: accel
3
4 integer:: nglob=TAILLE
5 integer:: dim=3
6 allocate(accel(nglob,dim))
7 end program exemple
```

Allocation mémoire en Fortran Pour permettre l'utilisation de la mémoire allouée via S_GPU 2 dans un programme Fortran, nous avons utilisé un module apparu avec la norme 2003 de ce langage, appelée `iso-c-binding`. Ce module fournit des fonctionnalités permettant d'utiliser une zone mémoire partagée allouée et de l'associer à un tableau Fortran. Nous illustrons la mise en place de notre approche par un exemple :

Exemple détaillé en Fortran Nous prenons ici l'exemple d'un programme Fortran qui souhaite utiliser S_GPU 2 pour exploiter des GPU. Ce programme doit envoyer certaines données sur le GPU pour faire des calculs. Nous allons montrer comment adapter ce programme pour qu'il puisse envoyer certaines de ses données sur un GPU.

Le programme original utilise `allocate` pour allouer des zones mémoire CPU. Pour adapter ce programme à S_GPU 2, les zones mémoire qui sont amenées à être transférées doivent utiliser l'allocateur de S_GPU 2.

Sur la figure 5.3, le programme Fortran original, déclare un tableau appelé `accel` (ligne 2) alloue de la mémoire grâce à la commande Fortran `allocate` (ligne 6). C'est ce tableau `accel` qui doit être envoyé sur le GPU.

La figure 5.4 montre les modifications à apporter pour utiliser notre allocateur. Une zone de mémoire partagée est allouée grâce à la fonction `allocate_cpu_mem_object` de l'API S_GPU 2. Cette zone mémoire doit être associée au tableau fortran `accel` que l'on souhaite envoyer sur le GPU. C'est le rôle de la fonction `c_f_pointeur` (ligne 8), qui "lie" un pointeur C à un pointeur Fortran. Cette fonction est disponible grâce au module `iso-c-binding`. Le tableau `accel` peut ensuite être utilisé comme un tableau fortran standard.

Cette procédure est compatible avec tous les compilateurs supportant la norme Fortran 2003 : c'est le cas des versions récentes d'Intel Ifort (v.10) et de GNU gFortran (v.4.4).

Une fois les zones de mémoire allouées et transférées, elles peuvent être utilisées dans le code des threads GPU qui sont soumis à notre bibliothèque.

5.3.2.2 Soumission et création des threads

Nous allons prendre un exemple, représenté sur les figures 5.6 et 5.5, pour montrer de manière détaillée comment soumettre des threads qui utilisent des zones de mémoire partagées et comment coder ces threads. L'exemple de code est issu de notre travail sur l'application Fortran SPECFEM3D qui sera couvert en détail au chapitre 8. Par souci de

Listing 5.4 – Code Fortran adapté pour S_GPU 2

```
1  program exemple
2    use iso_c_binding
3
4    real(kind=4), dimension(:,:),pointer :: accel
5    type(c_ptr) :: c_ptr
6
7    call  allocate_cpu_mem_object(sg2_cpu_accel,nglob*ndim*4,c_ptr)
8    call c_f_pointer(c_ptr, accel , [nglob,ndim])
9
10
11 end program exemple
```

clarté, nous représentons une soumission homogène qui comporte uniquement des threads GPU.

Soumission de threads Pour soumettre les threads GPU, l'application Fortran appelle la fonction `bind_callback_kernel1` (ligne 11, figure 5.5) qui va se charger de soumettre les threads à S_GPU 2. Cette fonction possède trois paramètres qui ont été initialisés par l'application Fortran. Ces trois paramètres correspondent à trois zones de mémoire partagées allouées par l'allocateur de S_GPU 2.

La fonction `bind_callback_kernel1` sérialise ces paramètres dans une instance de la structure `param_tk1` (lignes 15-17), cette structure contient donc les paramètres nécessaires au calcul. C'est le moyen que nous avons choisi pour le passage de paramètres, qui a été décrit en 5.1.2. Ensuite, une fonction de l'API S_GPU 2 `sg2_submit_threads_gpu` (ligne 19) est appelée : elle est l'équivalent de la fonction `soumettre_thread` du chapitre précédent (4.2.3.1). Le paramètre `bib_partagee.so` correspond au nom de la bibliothèque partagée dans laquelle se trouve la fonction `kernel1_grp`, qui est la fonction chargée de créer les threads GPU et qui sera détaillée quelque lignes plus bas. Il reste ensuite à passer les paramètres nécessaires au calcul, qui sont stockés dans la structure `param`.

Nous n'avons pas représenté de soumissions hybrides sur ce programme, mais il suffit d'ajouter des pointeurs vers les fonctions de coupe et de regroupement en paramètres lors de la soumission des threads.

Programmation des threads GPU L'exemple contenu dans 5.6, montre comment la programmation des threads GPU est effectuée. Tout le code présenté ci-après est compilé dans la bibliothèque partagée nommée `bib_partagee.so`. La fonction `kernel1_grp` (ligne 15), qui s'exécute donc dans l'espace d'adressage du processus S_GPU 2, reçoit dans l'argument `void* arg` une copie d'une structure de type `param_tk1`, qui a été créée lors de la soumission des threads (programme 5.5). Cette fonction peut ensuite créer un thread (`sg2_get_new_sg_tid()`) et y associer le code que l'on désire exécuter en tant que thread grâce à la fonction `sg2_depose_work_cuda` (ligne 24) de l'API S_GPU 2.

Listing 5.5 – Soumission d'un groupe de threads dans l'application

```
1
2  struct param_tk1
3  {
4    memory_id_shared_t displ;
5    memory_id_shared_t veloc;
6    memory_id_shared_t accel;
7  }
8
9
10 //------ Dans le code de l'application ------------------
11 void bind_callback_kernel1_(memory_id_shared_t** displ ,
       memory_id_shared_t** veloc,memory_id_shared_t** accel)
12 {
13   param_tk1 param;
14
15   param.d_displ = **displ;
16   param.d_veloc = **veloc;
17   param.d_accel = **accel;
18
19   accel_work_id_t wait = sg2_depose_threads_gpu("specfem_ker1",
20                                                 1,1,
21                                                 ``bib_partagee.so'',
22                                                 "kernel1_grp",
23                                                 &param,
24                                                 sizeof(param_tk1));
25
26
27   sg2_wait_for_work(wait);
28
29 }
```

Programmation des threads CPU Nous n'avons pas représenté, sur cet exemple, des threads CPU car contrairement aux threads GPU qui nécessitent des appels de fonctions S_GPU 2 spécifiques, les threads CPU se programment, comme nous l'avons déjà vu, en utilisant des environnements standards, comme OpenMP.

Listing 5.6 – Thread dans la bibliothèque partagée

```
1  //------ Dans la bibliotheque partagee ------------------
2  void pre_callback_UpdateDisplVeloc_sg2(info_accel_work* ia, void* arg)
3  {
4    param_tk1 *arg_ker1 = (param_tk1*)(arg);
5
6    .... code CUDA en utilisant les arguments de arg
7
8
9    kernel<<<ia->get_stream>>>(...);
10 }
11
12
13
14 extern "C"
15 void kernel1_grp(void *util,int N,void* arg)
16 {
17  // sg2_on_site_util *sg2_on = (sg2_on_site_util*)(util);
18
19   param_tk1 *arg_ker1 = (param_tk1*)(arg);
20
21   sgpu_thread_id sg_tid = sg2_get_new_sg_tid();
22
23
24   sg2_depose_work_cuda(sg_tid,&pre_callback_UpdateDisplVeloc_sg2,
         arg_ker1,sizeof(param_tk1));
25
26 }
```

5.4 Conclusion

Nous avons présenté S_GPU 1 et S_GPU 2, deux bibliothèques logicielles permettant de construire des applications basées sur les modèles définis dans le chapitre 4. Ces bibliothèques sont disponibles sur la forge ligforge : `http://sgpu.ligforge.imag.fr/`. La distribution de BigDFT contient également une archive de S_GPU 1.

Nous avons présenté le fonctionnement de nos deux bibliothèques et avons montré différents choix possibles d'implantation. Les points négatifs et positifs des différentes approches ont été abordés. Cela nous a permis de justifier les choix réalisés, comme l'architecture centralisée et l'utilisation de bibliothèques partagées.

Les deux bibliothèques doivent être vues comme des "preuves de concept". Elle ont été développées pour mettre en place rapidement certaines idées comme la virtualisation d'accélérateurs ou les calculs hybrides. Néanmoins, certaines approches, comme les calculs hybrides, pourraient être réutilisées dans des standards de type OpenMP ou OpenACC.

En donnant l'illusion à une application d'avoir autant d'accélérateurs que de tâches MPI, S_GPU 1 propose aux applications d'utiliser une configuration matérielle homogène. L'exécution hybride n'est cependant pas optimale : les exécutions simultanées CPU et GPU sont possibles mais difficilement contrôlables. Afin de permettre aux applications d'exploiter entièrement les ressources de calcul d'une grappe hybride, nous nous sommes basés sur S_GPU 1 et avons développé S_GPU 2. Grâce à la prise en charge des calculs hybrides, cette bibliothèque renforce les possibilités de parallélisme d'exécution entre CPU et GPU.

Nous avons pris soin à ce que nos bibliothèques définissent une interface de programmation exploitable avec C et Fortran afin d'être utilisables par une grande partie des applications scientifiques existantes. L'interopérabilité C – Fortran est donc un élément important puisque nos bibliothèques sont codées principalement en C/C++. Appeler des fonctions de notre API depuis un programme Fortran est assez aisé. Cependant, pour des utilisations avancées, comme allouer de la mémoire avec un allocateur spécifique codé en C, cela devient beaucoup plus difficile. Toutefois, la norme Fortran – notamment celle de 2003 –, évolue vers plus d'interopéablitilité avec C. Le module `iso_c_binding` par exemple, permet d'associer des pointeurs C avec des tableaux Fortran ou d'utiliser des type C directement dans un programme Fortran. Nous avons utilisé ces capacités d'interopéabilité de Fortran pour développer nos bibliothèques.

L'interopéablilité Fortran de nos deux bibliothèques pourrait toutefois être améliorée dans de futures versions. Au lieu d'appeler directement des fonctions C depuis Fortran, il pourrait être envisageable de créer une véritable interface Fortran en définissant le prototype des fonctions de l'API directement en Fortran. Cela permettrait au compilateur de faire certains contrôles sur les types directement à la phase de compilation. Pour le passage des paramètres, lors de l'appel de fonctions S_GPU 2, il serait intéressant de pouvoir sérialiser les paramètres depuis Fortran plutôt que de passer par une structure C.

Dans les chapitres suivants, nous allons évaluer S_GPU 1 et S_GPU 2, en les intégrant dans deux grandes applications scientifiques : BigDFT et SPECFEM3D.

Troisième partie

Évaluation des approches proposées

Chapitre Six

Description de BigDFT et SPECFEM3D

Sommaire

Jusqu'à présent, nous avons proposé des modèles de programmation mis en œuvre dans deux bibliothèques logicielles. Afin de rendre nos modèles intégrables dans des applications existantes, nous avons défini nos modèles comme des extensions de modèles de programmation existants.

Nos propositions étant destinées aux grandes applications, il est indispensable de les valider expérimentalement sur ce type d'applications. Ainsi, nous avons choisi de les intégrer dans deux logiciels de simulation scientifique existants : BigDFT et SPECFEM3D.

Il faut noter que nos deux bibliothèques ont été conçues en lien étroit avec les simulations scientifiques sur lesquelles elles ont été intégrées. Pour S_GPU 1, nous avons précisément évalué les besoins de BigDFT et avons conçu S_GPU 1, en conséquence. S_GPU 1 a donc été intégrée à BigDFT au fur et à mesure de sa mise au point. Nous avons utilisé la même démarche avec S_GPU 2 et SPECFEM3D. Cette démarche nous a permis d'avoir des bibliothèques adaptées aux applications scientifiques.

Nous décrivons, dans ce chapitre BigDFT et SPECFEM3D. Ces deux simulations scientifiques possèdent des caractéristiques intéressantes du point de vue informatique, présentées ici. Ce chapitre est organisé en deux points principaux, le premier porte sur BigDFT, le second sur SPECFEM3D.

6.1 Le logiciel de nanosimulation BigDFT

La première application scientifique que nous avons utilisée se nomme BigDFT. Il s'agit d'un logiciel libre [1] qui a été développé dans le cadre d'un projet européen entre 2005 et 2008. BigDFT a été codé en Fortran 90 et est relativement gros (plus de 70 000 lignes de code).

Le but de BigDFT est de calculer la structure électronique d'un système à l'échelle atomique en utilisant le formalisme de Kohn-Sham de la Théorie de la fonctionnelle de la densité DFT [2] [PTA+92]. Ce code appartient à la classe des simulations ab-initio, qui est une méthode largement utilisée pour calculer la structure et les propriétés des matériaux ou des molécules au niveau atomique. Le modèle physique de BigDFT se base sur des ondelettes de Daubechies lorsqu'il s'agit d'effectuer des opérations sur des fonctions d'ondes électroniques – ou orbitales – qui représentent les données physiques du problème. Les théories physiques utilisées et le fonctionnement du code de BigDFT sont détaillés précisément dans l'article de référence du projet [GNG+08].

Les ondelettes, qui sont au cœur de BigDFT, ont la particularité de permettre d'effectuer des opérations très localisées, ce qui est appréciable du point de vue informatique. En effet, des opérations localisées permettent de limiter les communications entre les nœuds de calcul d'une grappe et d'utiliser des mécanismes d'optimisation matériels, comme les caches, de manière efficace. BigDFT montre ainsi une scalabilité excellente sur les architectures de calcul parallèles, cette propriété est représentée par la figure 6.1. Cette figure montre l'efficacité parallèle du code pour plusieurs exécutions lorsque le nombre d'atomes simulés diffère. À côté de chaque point, le nombre d'orbitales (ou fonction d'onde) est indiqué.

BigDFT se base sur un algorithme itératif : un ensemble de traitements identiques est successivement exécuté. Les itérations sont terminées lorsque le résultat a convergé. Bien que la nature et le nombre de ces traitements varient en fonction du système utilisé, on peut isoler deux principales opérations mathématiques largement utilisées dans BigDFT :

— Un ensemble de convolutions tridimensionnelles avec des filtres petits et séparables.
— Des fonctions d'algèbre linéaire, effectuées grâce à des appels de fonction BLAS [LHKK79].

Ces deux opérations sont exécutées de manière intensive pour la grande majorité des parties du code. Le ratio de calcul entre les convolutions et les autres opérations dépend fortement du système atomique que l'on considère, mais pour la plupart des systèmes, les convolutions prennent plus de 50 % du temps total de calcul.

Les opérations de convolution et d'algèbre linéaire possèdent de bonnes propriétés pour envisager de les paralléliser : nous verrons un peu plus loin que ce sont ces opérations qui ont été choisies pour être déportées sur GPU.

6.1.1 Parallélisation de BigDFT

BigDFT est un code SPMD homogène qui utilise MPI pour sa parallélisation. BigDFT ne nécessite pas de partitionneur externe pour paralléliser les données du problème car il

1. Sous licence GNU-GPL
2. Density Functional Theory, ou *Théorie de la fonctionnelle de la densité*

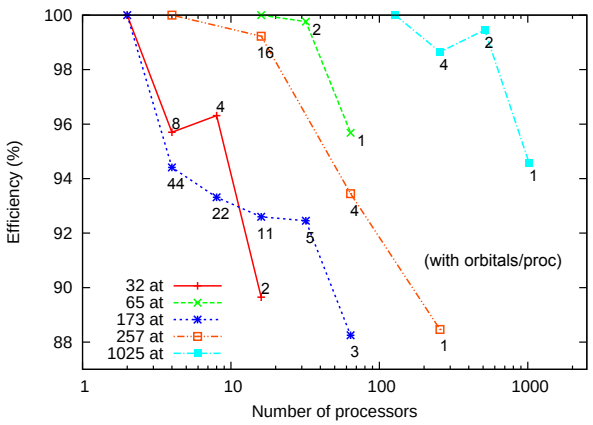


Figure 6.1 – Efficacité de la parallélisation de BigDFT

calcule, lors de sa phase d'initialisation, la répartition des orbitales entre les tâches MPI.

6.1.1.1 Répartition des orbitales dans les tâches MPI

BigDFT étant un programme SPMD homogène, toutes les tâches MPI font exactement les mêmes opérations, ce sont les données – les orbitales pour BigDFT – que l'on va partager. Lorsque l'on fait une simulation sur un système physique, ce système possède un certain nombre d'orbitales n_{orbs}. Supposons que lors d'une exécution, BigDFT crée n_{MPI} tâches MPI. Sans manipulation supplémentaire, le nombre d'orbitales traité par chaque processus est : n_{orbs}/n_{MPI}. On appelle cette répartition, « répartition homogène ».

On peut également avoir une répartition non-homogène, dans ce cas, il est possible de spécifier précisément le nombre d'orbitales qu'un processus MPI particulier va traiter.

6.1.1.2 Parallélisation du traitement des orbitales

Il y a principalement deux phases de traitement dans BigDFT. Dans la première, chaque fonction d'onde est traitée de manière indépendante : il est ainsi possible de traiter plusieurs fonctions d'onde simultanément. Dans BigDFT, chaque processeur d'une grappe peut donc traiter indépendemment les fonctions d'onde, ce qui entraîne une possibilité élevée de parallélisation du code. Le maximum de processeurs utilisables correspond au nombre de fonctions d'onde du système (de l'ordre de plusieurs milliers).

Durant la deuxième phase, une matrice dont les coefficients sont distribués parmi tous les processeurs doit être transposée. Des communications MPI sont donc lancées pour assurer cette transposition.

6.1.1.3 Communication MPI

La transposition de la matrice a été mise en place dans BigDFT avec l'aide de communications collectives MPI (de type `ALLTOALL`). Ce choix peut s'avérer problématique dans les très grands centres de calcul possédant plusieurs dizaines de milliers de nœuds où les communications collectives peuvent devenir très coûteuses. Malgré cela, sur des

architectures matérielles à réseaux performants, BigDFT montre une bonne scalabilité et confirme les résultats de la figure 6.1.

6.1.2 Utilisation des GPU par BigDFT

Les convolutions et les routines d'algèbre linéaire étant les deux principales opérations exécutées par BigDFT, ce sont ces deux-là qui ont été portées sur le GPU.

NVIDIA fournit une bibliothèque très performante pour l'algèbre linaire, appelée CUBLAS, qui a été utilisée directement par BigDFT. Pour les convolutions, des optimisations spécifiques ont été réalisées par l'équipe développant de BigDFT.

6.1.2.1 Produit de convolutions périodiques en trois dimensions

Trois types de convolutions sont utilisés dans BigDFT. Elles sont toutes similaires. Pour fixer les idées nous décrivons la mise en œuvre des convolutions 3D périodiques dans BigDFT . Elles se basent sur trois appels successifs de convolution unidimensionnelle.

Algorithme périodique unidimensionnel La convolution discrète est l'opération mathématique suivante :

$$y(t) = \sum_{k=-n1}^{n2} x(k)h(t-k)$$

$h(t)$ représente le filtre, et $x(t)$ la ligne sur laquelle on applique le filtre pour obtenir une nouvelle ligne $y(t)$

La taille du filtre, égale à $n2 - n1$, est assez courte. Typiquement, $n2 - n1 < 30$ dans BigDFT.

Le produit de convolution périodique est un produit de convolution classique dans lequel le filtre reprend les éléments du début de la ligne lorsqu'il arrive à la fin de celle-ci. On représente cela sur la figure 6.2.

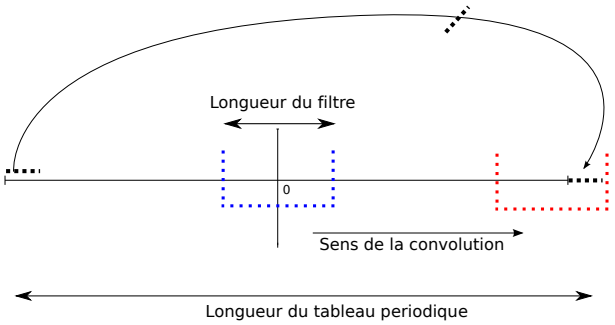


Figure 6.2 – Une convolution 1D périodique

De manière séquentielle, la convolution s'écrit très simplement avec deux boucles imbriquées.

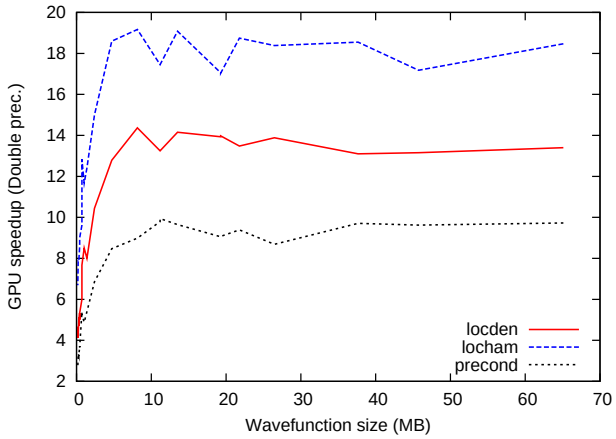


Figure 6.3 – Performance des convolutions de BigDFT en CUDA

Extension à trois dimensions de l'algorithme périodique unidimensionnel Il suffit pour cela d'appliquer la convolution périodique précédente sur chacune des trois dimensions de la matrice.

Soit `A` une matrice de dimensions (x, y, z), représentée par le tableau `T` de taille $x \times y \times z$. Pour obtenir la matrice `R` résultat, qui sera également de dimensions (x, y, z), nous appliquons trois fois l'algorithme sur `T`, en considérant tout d'abord ce tableau comme une matrice 2D de dimensions $(x \times y, z)$, puis de dimensions $(x, y \times z)$ et enfin $(z \times x, y)$. Après ces trois étapes, le tableau `T` vu en 3D représente la matrice `R`.

Mise en œuvre en CUDA des convolutions Chaque colonne de la matrice étant indépendante, il est possible de créer un grand nombre de threads CUDA traitant chacun une ligne de manière indépendante. Cette stratégie exhibe de très bonnes performances sur les GPU NVIDIA. Nous pouvons apprécier, sur la figure 6.3, que par rapport à un cœur CPU, une accélération comprise entre 10 est 19 est atteinte pour les trois types de convolution. Il faut bien noter que c'est la double précision qui est ici utilisée.

BigDFT a l'avantage de nécessiter très peu de communications entre la mémoire CPU et la mémoire GPU : les données peuvent rester sur le GPU ce qui améliore ainsi les performances en évitant les transferts mémoire.

6.1.3 Communication MPI et calcul GPU

BigDFT est un code MPI qui utilise des GPU ; il faut donc gérer une cohabitation entre l'environnement CUDA et l'environnement MPI. Le code peut utiliser autant de tâches MPI que de cœurs CPU présents dans la grappe sur laquelle le code s'exécute si le système physique à simuler contient au moins autant d'orbitales que de cœurs CPU.

Dans la version de BigDFT que nous avons utilisée, chacune des tâches MPI peut utiliser un GPU pour exécuter certaines parties des calculs. Il est également possible de spécifier quelles tâches MPI utilisent un GPU et quelles tâches utilisent uniquement un CPU. Ainsi, nous avons un contrôle sur l'utilisation des GPU par BigDFT.

La liste suivante montre les opérations effectuées simultanément par toutes les tâches

MPI lors d'une itération de BigDFT. Cette liste permet d'identifier à quels moments une communication est effectuée et à quels moments un GPU peut être utilisé.

1. Construction de la densité éléctrique à partir des fonctions d'ondes (**) ;
2. Solveur de l'équation de Poisson (**) ;
3. Application de l'hamiltonien local (**) ;
4. Application de l'hamiltonien non-local ;
5. Préconditionneur (**) ;
6. Mise à jour des fonctions d'ondes ;
7. Orthogonalisation des fonctions d'ondes.

Entre les étapes 4 et 5, 5 et 6 et 6 et 7, ont lieu des communications MPI.

Les étapes marquées de (**) exploitent le GPU, ce sont principalement les calculs de convolutions, utilisant le code CUDA décrit précédemment, et les calculs d'algèbre linaire – utilisant la bibliothèque NVIDIA CUBLAS –.

Il faut bien noter que, lors d'une itération, seule une partie des calculs peut se faire sur un GPU. Le reste s'effectue sur le CPU. BigDFT est donc un code hybride dans le sens ou lors d'une même itération, des calcul exploitent les CPU et d'autres les GPU.

6.1.4 Conclusion

L'utilisation des GPU dans BigDFT utilise une approche hybride : certains calculs restent sur le CPU et d'autres sont sur le GPU. Cette approche a été choisie car BigDFT ne possède pas de partie centrale réunissant la quasi totalité du temps de calcul, mais les calculs sont répartis dans tout le code.

Nous avons ici présenté de manière assez synthétique le code de BigDFT tel qu'il était fin 2008. Le portage des convolutions vers les GPU venait d'être partiellement terminé et l'objectif était de déterminer le meilleur moyen d'exploitation des grappes de calcul hybrides pour ce code. Toutes nos expérimentations se basent sur cette version du code.

Aujourd'hui, BigDFT a évolué, des travaux ont permis l'utilisation d'OpenMP en plus de MPI, les convolutions ont été converties en OpenCL et de gros travaux sont effectués afin de chercher à diminuer le poids des communications MPI.

6.2 Le simulateur d'ondes sismiques SPECFEM3D

Le deuxième code que nous avons utilisé est SPECFEM3D [TKL08] qui est un code qui permet de simuler la propagation d'ondes sismiques ; il est basé sur la méthode mathématique des éléments finis. Cette application scientifique permet de simuler avec une très grande précision la propagation des ondes sismiques terrestres. Ce niveau de précision implique que les quantités de données manipulées par SPECFEM3D ainsi que les calculs sont très souvent conséquents. SPECFEM3D produit un sismogramme qui montre l'état des ondes sismiques en fonction du temps. Ce sismogramme peut être interprété : la figure 6.4 montre un exemple de résultat, un séisme ayant eu lieu en Chine a été "rejoué" par la simulation grâce à SPECFEM3D.

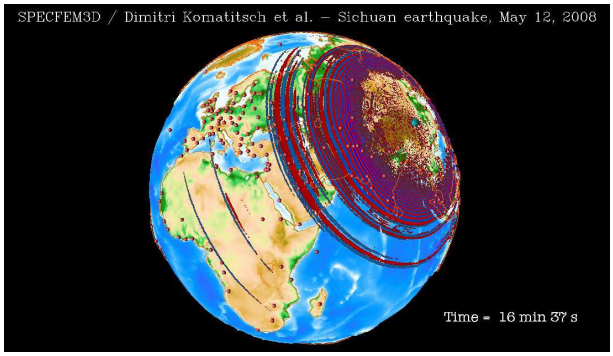


Figure 6.4 – Capture d'écran d'une simulation d'un séisme ayant eu lieu dans la province chinoise du Sichuan, Mai 2008.

SPECFEM3D est un code complexe comportant plusieurs milliers de lignes de code. Comme BigDFT, il est lui aussi codé avec le langage Fortran. Contrairement à BigDFT, où il est difficile de trouver une partie de code unique concentrant une grande partie des besoins en calcul, la plupart des besoins en puissance de SEPCFEM3D est contenue dans une partie du code appelée la boucle en temps.

6.2.1 Parallélisation du code

La parallélisation de SPECFEM3D est basée sur une approche SPMD et utilise MPI. Chaque tâche MPI traite une partie d'un maillage généré par un mailleur externe. Les communications peuvent rester relativement locales : chaque tâche MPI ne communique qu'avec ses voisins directs. SPECFEM3D est ainsi bien adapté à une parallélisation massive sur de grandes grappes de calcul. Le code a d'ailleurs gagné le prix Gordon Bell [CKL+08] grâce à sa capacité à exploiter des grappes comportant plusieurs dizaines de milliers de processeurs.

6.2.2 Mise en œuvre de l'exploitation des GPU

Dans SPECFEM3D, une boucle qui met en œuvre une boucle en temps concentre la plupart du temps de calcul d'une simulation. Pour accélérer le code de SPECFEM3D, les auteurs du code ont donc porté entièrement cette boucle en temps sur des GPU.

Un article [KEGM10] présente de manière précise la manière avec laquelle SPECFEM3D a été porté sur GPU. Des adaptations, décrites brièvement ci-après, dans l'algorithme principal de la boucle ont dû être apportées afin de permettre une parallélisation massive des traitements pour exploiter efficacement les GPU.

Nous détaillons ici la version GPU de la boucle en temps. Cette boucle, représentée sur la figure 6.6, comprend trois parties principales. Les parties 1 et 3 mettent à jour les vecteurs globaux de la simulation. La deuxième partie de la boucle en temps calcule les vecteurs des force élastiques dans chaque élément spectral du maillage. Ces vecteurs de forces sont ensuite sommés pour chaque élément commun du maillage. Nous verrons un peu plus loin comment cette somme a été adaptée pour pouvoir exploiter des technologies manycores comme les GPU.

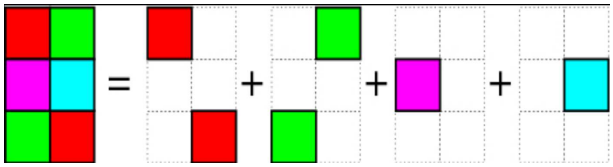


Figure 6.5 – Découpage du traitement en différentes couleurs (© Dimitri Komatitsch)

Au niveau du code source, les parties 1 et 3 contiennent un kernel CUDA alors que la partie 2 contient une succession de n appels au même kernel CUDA. (n dépend du maillage utilisé mais il vaut généralement entre 15 et 30). Des dépendances de données imposent que les trois parties soient exécutées successivement. Tous les calculs sont réalisés en simple précision et peuvent donc ainsi utiliser toute la puissance des accélérateurs de type GPU.

Cette boucle est exécutée dans chaque tâche MPI et traite trois vecteurs différents :

— `accel` : représentant des informations d'accélération
— `veloc` : représentant des informations de vitesse
— `displ` : représentant des informations de déplacement

Les parties 1 et 3 mettent en œuvre des sommes et des multiplications entre les éléments des trois tableaux, ce sont des opérations très parallélisables et bien adaptées à un portage sur GPU.

Pour la partie 2, c'est le tableau `accel` qui est modifié. Néanmoins, différents points du tableau `accel` sont généralement modifiés plusieurs fois au cours du calcul de cette partie. Cela ne pose pas de problème dans une version séquentielle, mais une version massivement parallèle sur GPU imposerait de mettre en place des dispositifs coûteux de synchronisation si au même moment, deux threads CUDA souhaitaient modifier un même élément du tableau. Pour résoudre ce problème, les concepteurs de SPECFEM3D ont découpé le traitement de cette partie 2 grâce à un mécanisme de coloriage. Ce coloriage, calculé directement durant la phase de maillage du système est décrit sur la figure 6.5. Les cases de même couleur contiennent des éléments n'ayant pas de points communs. Il est donc possible de traiter en parallèle tous les éléments d'une même couleur sans se soucier d'un éventuel conflit d'accès sur des ressources. Cette approche élimine donc le besoin d'utiliser des mécanismes de synchronisation et peuvent utiliser pleinement les ressources parallèles offertes par les GPU. Chaque kernel de la partie 2 s'occupe donc du traitement d'une couleur.

Communications CPU <-> GPU Dans SPECFEM3D, quasiment toutes les données peuvent être conservées sur le GPU durant une exécution, les quelques rares transferts CPU <-> GPU, permettant de mettre à jour le sismogramme résultat et d'alimenter les communications MPI. Ces communications sont très réduites et donc peu coûteuses.

Une réécriture en C Le code de calcul de SPECFEM3D a été partiellement réécrit en C par les auteurs lors de la conception du code GPU pour en simplifier le portage. L'équipe MAGIC3D de l'INRIA a, au début de l'année 2010, repris le code Fortran de SPECFEM3D et l'a adapté pour utiliser les GPU avec notre bibliothèque S_GPU 1. Ce

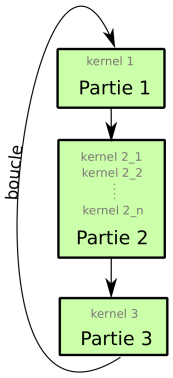


Figure 6.6 – Boucle principale de calcul

passage en Fortran est important car il permet d'intégrer la version accélérée du code dans la totalité du code de SPECFEM3D. C'est à partir du code Fortran que nous avons expérimenté dans le chapitre 8.

6.2.3 Communications MPI et calcul GPU

Les appels de kernel de la partie 2 peuvent être scindés en deux groupes. Le premier groupe s'occupe des éléments situés au bord de la maille traitée par la tâche MPI et les autres les éléments du centre de la maille. Sachant que chaque tâche MPI communique à son voisin uniquement les données qui ont été modifiées sur les bords, il est possible, après l'exécution du premier groupe, de démarrer des communications MPI asynchrones s'exécutant en même temps que les calculs du deuxième groupe. Ces communications asynchrones permettent de recouvrir le temps de communication par du calcul et augmentent encore les performances du code.

Comme les CPU ne sont pas utilisés pour faire des calculs, le nombre de tâches MPI lancé est égale au nombre de GPU dans la machine. Ainsi, chaque tâche MPI utilise de manière unique un GPU. Sur les grappes actuelles équipées de plusieurs dizaines de cœurs par nœud de calcul, SPECFEM3D laisse ces cœurs inutilisés et se prive ainsi d'une grande partie de la puissance de calcul.

6.2.4 Conclusion

La stratégie d'exploitation des GPU par SPECFEM3D se base sur l'utilisation exclusive des GPU dans les phases de calcul intensives. Il n'y a donc pas utilisation des cœurs CPU pour le calcul. Le travail effectué sur ce code pour le porter sur GPU est conséquent : un nouvel algorithme, basé sur une coloration a été mis en place pour exploiter efficacement des GPU. SPECFEM3D représente donc un code qui n'était pas naturellement massivement parallélisable pour GPU et a dû être adapté en conséquence.

Le code de SPECFEM3D hybride sur lequel nous avons expérimenté était déjà un code de production utilisé sur des grappes comportant plusieurs centaines de GPU. Il n'était donc pas pertinent d'essayer d'optimiser ce code pour sa partie GPU. Les expériences, relatées au chapitre 8, montrent comment nous avons proposé d'étendre ce code pour qu'il exploite simultanément CPU et GPU.

6.3 Conclusion

Les deux applications sur lesquelles nous avons choisi d'expérimenter sont assez caractéristiques des applications scientifiques disponibles aujourd'hui :

— elles sont scalables ;
— elles sont relativement grosses (plusieurs dizaines de milliers de lignes de codes) ;
— elles utilisent un modèle SPMD et sont basées sur MPI.

Les deux applications sont utilisées comme des benchmarks dans différents centre de recherches et notamment dans le projet européen PRACE.

Dans BigDFT, on trouve une succession de parties accélérées et de parties CPU. Ainsi, lors de l'exécution de la version GPU de BigDFT, les CPU continuent à être utilisés. À l'inverse, pour SPECFEM3D, une partie accéléré concentre l'essentiel du temps d'exécution : seuls les GPU sont utilisés pour l'essentiel du déroulement de SPECFEM3D.

Pour nos expérimentations, nous avons pris les deux codes à des stades différents de leurs développements. La version de SPECFEM3D sur laquelle nous avons travaillé avait déjà été largement utilisé en production sur des grappes de calcul hybrides – nous avons commencé les travaux fin 2010 –. La version de BigDFT, quant à elle, était à un stade préliminaire d'exploitation des GPU – travaux commencés fin 2008 –. Comme pour le matériel hybride, les applications hybrides évoluent rapidement et nous avons donc utilisé les versions des codes disponibles au début de nos expérimentations.

La validation expérimentale de nos modèles et bibliothèques logicielles se base sur ces deux simulations scientifiques. La virtualisation d'accélérateurs, permettant de donner l'illusion d'un nombre identique de CPU et d'accélérateurs est bien adapté à un code hybride comme BigDFT qui utilise CPU et GPU dans ses calculs ; le chapitre suivant propose donc de montrer l'intérêt de S_GPU 1 avec BigDFT. SPECFEM3D utilise uniquement les GPU pour ses calculs. Il y aurait peu d'intérêt à utiliser S_GPU 1 pour ce code car les CPU ne sont pas exploités. En revanche, nous avons adapté SPECFEM3D pour qu'il lance des calculs hybrides, avec l'aide de S_GPU 2. Ces expériences sont décrites au chapitre 8.

Chapitre Sept

Étude de S_GPU 1 avec BigDFT

Sommaire

Nous allons maintenant évaluer l'intérêt de la virtualisation de GPU en comparant deux modèles d'exploitation hybrides avec l'application BigDFT [OGD09, GOD+09, GOV+11, GVO+11]. Le premier consiste à utiliser la virtualisation : chaque tâche MPI a l'illusion d'avoir les mêmes ressources de calcul CPU et GPU ; chacune d'entre elles pourra exploiter une puissance de calcul identique. Pour l'autre approche, la virtualisation n'est pas utilisée : certaines tâches MPI utiliseront des GPU et des CPU et d'autres uniquement des CPU. Les tâches MPI seront ainsi plus ou moins puissantes en fonction des ressources de calcul utilisées.

Après avoir montré comment S_GPU 1 a été intégré dans BigDFT, nous présenterons les performances des deux modèles d'exploitation hybrides, tout d'abord sur un seul nœud, puis sur une grappe hybride. Nous finirons ce chapitre par une évaluation de l'intérêt des GPU concernant la consommation énergétique.

7.1 Mise en place de S_GPU 1 dans BigDFT

BigDFT est un code dont certaines sections utilisent la ressource CPU et d'autres utilisent la ressource GPU. La version de BigDFT que nous avons employée avait un support préliminaire des GPU et utilisait, par exemple, uniquement un cœur CPU et un GPU sur des machines multicœurs multi-GPU. Pour sortir de ce mode 1 – 1, et ainsi pouvoir utiliser la totalité des ressources de calcul disponibles, nous avons décidé de mettre en place S_GPU 1 dans BigDFT afin que chaque tâche MPI puisse utiliser un GPU virtuel.

Nous avons conçu S_GPU 1 simultanément avec son intégration dans BigDFT. Il était donc possible d'adapter l'interface de programmation de S_GPU 1 aux besoins de BigDFT. Ainsi, nous avons progressivement augmenté l'interface Fortran de notre bibliothèque pour la rendre efficace dans BigDFT. Nous avons cependant pris soin à ce que S_GPU 1 soit totalement indépendante de BigDFT.

Dans le chapitre précédent, nous avons listé les sections du code de BigDFT qui utilisent les GPU. Nous allons prendre l'exemple du préconditionneur. Dans la version initiale de BigDFT, des transferts mémoire étaient effectués, suivis de l'appel du kernel CUDA implantant le préconditionneur. Cette organisation est décrite sur le premier exemple de code ci-dessous :

```
1        do iorb=1,num_orbs
2         call cuda_memcpy(tab_orbitales(i),...) !transfert des orbitales
              vers le GPU
3        end do
4
5        call kernel_preconditionneur(...)  !execution du kernel
6
7        call cuda_memcpy(result_precond,...)  !retour des resultats
             depuis le GPU
```

Pour la version S_GPU 1 de BigDFT, nous avons modifié légèrement le code afin de créer un thread S_GPU 1 qui exécute les transferts mémoire et lance l'appel au kernel CUDA. La structure initiale du code n'a donc pas du tout été modifiée. Cette version est représenté sur le deuxième exemple ci-dessous :

```
1       call sg_create_thread(thread_ptr) !creation du thread sgpu1
2
3       do iorb=1,num_orbs
4          call sg_gpu_send(tab_orbitales(i),...,thread_ptr)  !ajout du
               transfert des orbitales vers le GPU dans le thread
5       end do
6
7       call calcul(preconditionneur,...,thread_ptr)  !ajout de l'
            execution du kernel dans le thread
8
9       call sg_gpu_recv(result_precond,..., thread_ptr)  ! ajout du
            retour des resultats dans thread
10
11
12      call sg_launch_all_threads()  !lancement du thread cree
```

La même approche a été appliquée pour toutes les parties GPU du code de BigDFT.

S_GPU 1 est intégré dans la distribution standard de BigDFT, qui a été déployée en production sur les grappes de calcul. Il n'y a pas une version spécifique de BigDFT pour GPU et une autre pour CPU. Le même code contient et la version CPU, et la version GPU des calculs. Il est ainsi possible de décider au démarrage de BigDFT, grâce à un fichier de configuration, si c'est la version accélérée ou la version GPU qui sera utilisée. Cette configuration peut se faire au niveau de la tâche MPI. Ainsi, il est tout à fait

envisageable d'avoir une exécution de BigDFT où certaines tâches utilisent des GPU et d'autres uniquement des CPU.

La grande flexibilité du code de BigDFT rend donc possible d'adapter, au niveau de la tâche MPI, les ressources de calcul utilisées ainsi que la répartition de données à traiter. Nous avons exploité cette souplesse pour mettre au point différentes expériences relatées dans la suite de ce chapitre.

7.2 Évaluation de la virtualisation des GPU

BigDFT est un code dont certaines sections utilisent les CPU et d'autres utilisent les GPU. Ainsi, BigDFT peut utiliser la totalité des ressources de calcul d'une machine (CPU et GPU) lors d'une exécution.

L'objectif de cette section est de comprendre quel est le meilleur modèle d'exploitation hybride à utiliser dans BigDFT. Nous utilisons deux modèles d'exploitation, décrits ci-après, qui ont la capacité d'exploiter les ressources CPU et GPU d'une machine.

Les deux modèles évalués sont décrits ci-après :

Virtualisation des GPU Nous utilisons ici la virtualisation d'accélérateurs. Chaque tâche MPI a une puissance identique et la répartition des orbitales (données) est régulière (homogène). Cette répartition est représentée sur la figure 7.1. Chaque tâche MPI s'exécute sur un cœur et toutes les tâches MPI ont le même volume de calcul à effectuer. Toutes les tâches MPI du programme accèdent donc à un GPU virtuel contrôlé par S_GPU 1.

Pour évaluer l'intérêt de ce modèle, nous utilisons l'application BigDFT associée à notre bibliothèque S_GPU 1.

Sans virtualisation Dans ce modèle, il n'y a plus de virtualisation. Pour utiliser toutes les ressources de calcul, certaines tâches MPI utilisent les accélérateurs, et d'autres n'en utilisent pas. Il y a autant de tâches accélérées que de GPU. La répartition des orbitales est non-homogène sur les processus MPI . Cette répartition est représentée sur la figure 7.2 : la tâche 1 utilise un GPU et traite donc plus d'orbitales que les autres qui n'en utilisent pas.

Les tâches MPI qui ont le privilège d'utiliser un GPU disposent donc d'une capacité de calcul plus importante. Pour ces tâches privilégiées, le nombre d'orbitales qui leur est assigné est donc plus important.

Pour évaluer l'intérêt de ce modèle, nous utilisons encore une fois le code de BigDFT associé à S_GPU 1. Cependant, S_GPU 1 a été modifié : nous avons retiré l'attribution automatique des GPU aux tâches MPI ainsi que l'emploi des sémaphores.

Après avoir présenté les différentes zones du code de BigDFT qui nous ont servi à mesurer les performances de l'application, nous évaluerons les deux modèles d'exploitation présentés.

7.2.1 Indicateurs de performances utilisés

Afin de comprendre le comportement de BigDFT en fonction du mode d'exploitation choisi, nous avons étudié trois zones du code de BigDFT qui nous semblaient importantes.

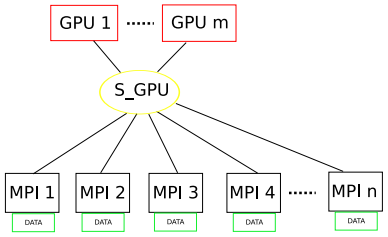


Figure 7.1 – Répartition homogène des données (Avec virtualisation des GPU).

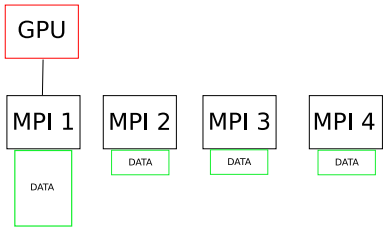


Figure 7.2 – Répartition non homogène des données (Sans virtualisation des GPU).

Ces trois zones sont les suivantes :

TOTAL Représente le temps total d'exécution des itérations de BigDFT. Il s'agit donc ici du temps d'exécution du cœur du calcul, les phases d'initialisation ne sont pas prises en compte.

PRECOND Est le temps d'exécution du préconditionneur de BigDFT. Comme vu en 6.1.3, c'est un temps intéressant car le préconditionneur est entièrement porté sur le GPU. Cela nous permettra de voir précisément l'impact des GPU sur les performances du code.

APPLY_PROJ Correspond au temps d'exécution de la partie du code chargé de l'application des projecteurs. Dans la liste en 6.1.3, cela représente l'application de l'hamiltonien non-local. Cette partie n'exploite donc que les CPU.

Pour enregistrer les temps d'exécution de ces trois zones, nous avons instrumenté BigDFT avec ITAC (Intel Trace Analyser and collector). ITAC nous permet d'analyser ces temps pour chaque tâche MPI. Cette caractéristique est importante car lors de nos expériences, les différentes tâches MPI ne vont pas traiter chacune la même quantité de données et ne vont pas utiliser les mêmes ressources de calcul. Par conséquent, il nous faut étudier des temps d'exécution par tâche, et non pas globalement pour l'ensemble de l'application.

7.2.2 Contexte expérimental

Nous avons utilisé deux machines lors de nos expériences. La première est basée sur deux processeurs INTEL Xeon X5570 (Nehalem) avec 24 Go de mémoire. Cette machine est connectée, via PCI Express, à un "cabinet Tesla" S1070 contenant quatre GPU. Ces GPU ont chacun 4 Go de mémoire et 240 cœurs. Dans nos expériences, nous utilisons uniquement deux GPU du cabinet Tesla. La deuxième machine est similaire mais utilise un processeur L5420 (Harpertown), moins performant que le X5570.

Processeur	**Xeon X5570 (Nehalem)**			**Xeon L5420 (Harpertown)**		
Nombre de tâches	1	4	8	1	4	8
TOTAL						
0 GPU (s)	573.00	160.00	91.00	892.00	272.00	214.00
1 GPU (s)	x	94.00	90.00	x	114,8	109.00
2 GPU (s)	x	55.00	50.00	x	73.00	58.00

(a) Temps d'exécution en seconde de BigDFT

Processeur	**Xeon X5570**	**Xeon L5420**
Nombre de tâches	8	8
TOTAL		
0 GPU (accel)	1	1
1 GPU (accel)	1.01	1.96
2 GPU (accel)	1.82	3.69

(b) Accélération, lorsque 8 tâches MPI sont utilisées. La référence est 8 tâches MPI n'utilisant pas de GPU

Table 7.1 – Temps et accélération pour une répartition homogène des données avec deux processeurs différents. La virtualisation est utilisée : les tâches MPI partagent les GPU avec S_GPU 1

Nombre de tâches	1	4	8
PRECOND			
1 GPU (s)	x	14.90	14.92
2 GPU (s)	x	7.35	7.40

(a) Temps d'exécution en seconde du préconditionneur

Nombre de tâches	1	4	8
PRECOND			
1 GPU (accel)	x	1	1
2 GPU (accel)	x	2,03	2,02

(b) Accélération, la référence est 4 tâches MPI utilisant 1 GPU

Table 7.2 – Temps et accélération du préconditionneur sur GPU

Concernant les CPU, le L5420 est moins puissant que le X5570 ce qui permet d'étudier l'impact de la puissance des CPU sur les performances hybrides.

7.2.3 Approche 1 : Virtualisation des GPU

Nous commençons nos expériences en évaluant l'intérêt de la virtualisation d'accélérateurs sur un seul nœud hybride. Chaque tâche MPI a l'illusion d'avoir la même puissance de calcul et la répartition des données peut donc être homogène. Chaque tâche MPI traite le même nombre d'orbitales et nous étudions ici les temps "TOTAL" et "PRECOND".

Les résultats sont représentés à l'aide de tableaux. Sur les tableaux 7.1, on montre le temps d'exécution et le ratio d'accélération de TOTAL lorsque le nombre de GPU et de tâches MPI varient.

Les tableaux 7.2 quant à eux, montrent les temps d'exécution et les accélérations pour l'indicateur PRECOND.

Observations et analyse :

Coût du partage Pour partager un GPU, S_GPU 1 sérialise les demandes d'accès aux GPU provenant des différentes tâches MPI. Cette sérialisation ne diminue pas les performances du code GPU de BigDFT. En effet, en s'aidant du premier tableau 7.2, "PRECOND" dure 15 s pour 4 tâches MPI (il y a donc un ensemble de 4 kernels sérialisés), et dure également 15 s pour 8 tâches MPI (il y a pourtant deux fois plus de kernels sérialisés). Les kernels de BigDFT sont d'un grain relativement gros (14s). Les appels système engendrés par l'utilisation, qui sont de l'ordre de la milliseconde, n'ont donc aucune influence sur les performances du code.

Utilisation de plusieurs GPU Notre bibliothèque autorise simultanément l'utilisation de plusieurs GPU. Pour cette raison, lorsque l'on passe de 1 à 2 GPU, le temps d'exécution de PRECOND est réduit d'un facteur 2 (accélération de 2 sur le tableau 7.2.b.) car les deux GPU travaillent en même temps.

Impact du ratio de performance CPU / GPU Le tableau 7.1.a indique que lorsque les huit CPU sont utilisés avec 8 tâches MPI, le temps d'exécution de la version CPU de BigDFT est similaire au temps de la version GPU sur la machine à base de X5570. Il faut donc utiliser un deuxième GPU pour avoir de meilleures performances sur cette machine.

Le ratio de performances, montré sur le tableau 7.1.b, entre les GPU et les CPU est important ici ; avec un processeur de type harpertown, beaucoup moins puissant que le Nehalem, l'ajout des GPU permet de gagner un facteur 3.69 (lorsque la totalité des cœurs CPU et GPU est utilisé), contre 1.82 lorsque le processeur est du type X5570. Le ratio de performances entre les CPU et les GPU doit donc être pris en compte lors d'étude de performances hybrides, les comparaisons les plus pertinentes étant de comparer des architectures de même générations : par exemple un Nehalem avec un TESLA C1070. Comparer une carte GPU récente avec un processeur plus ancien augmente mécaniquement l'accélération du code GPU par rapport au code CPU.

Avec des processeurs de type Nehalem et des cartes graphiques de type TESLA, par rapport à une exécution où les 8 cœurs du nœud sont utilisés, l'utilisation de deux GPU avec S_GPU 1 permet de doubler les performances en calcul de BigDFT. En donnant l'illusion à chaque tâche MPI d'avoir un GPU, la virtualisation permet de conserver une répartition de données identique entre la version hybride et CPU de BigDFT. Néanmoins, avec des codes flexibles, comme celui de BigDFT, il est possible de changer assez facilement la répartition des données.

Il s'agira donc, dans la suite de ce chapitre, de comparer notre approche basée sur la virtualisation à une approche où les données ne sont plus réparties équitablement entre les tâches MPI.

7.2.4 Approche 2 : Sans virtualisation d'accélérateurs

Nous explorons ici une autre voie d'utilisation des GPU pour laquelle il n'y a plus de virtualisation. Pour utiliser encore une fois toutes les ressources CPU et GPU disponibles,

nous configurons les ressources utilisées par les tâches MPI en fonction de la configuration matérielle du nœud. Nous définissons deux groupes de tâches MPI : le premier groupe, ou groupe GPU, contient les tâches utilisant les GPU et les CPU. Dans l'autre groupe, les tâches MPI utilisent uniquement les CPU. Il y a autant de tâches dans le groupe GPU que de GPU physiques disponibles. Nous appellerons dans cette partie « tâches GPU » les tâches MPI du groupe GPU.

Contrairement aux expériences précédentes, le nombre d'orbitales doit être adapté pour les différentes tâches : les tâches GPU sont les plus puissantes, elle ont plus d'orbitales. Dans le cas contraire, et parce que BigDFT est un code fortement synchronisé, si toutes les tâches avaient le même nombre d'orbitales, les tâches GPU traiteraient leurs données beaucoup plus rapidement que les tâches CPU et elles devraient attendre la fin d'exécution CPU ; ce qui annulerait l'avantage des GPU. Nous avons déterminé la répartition optimale des orbitales afin d'avoir un temps de calcul minimum. En nous aidant encore une fois de l'outil Intel Trace Analyser And Collector (ITAC), nous avons, de manière empirique, déterminé quelle était la meilleure répartition – c'est-à-dire celle qui donne un temps TOTAL minimal –.

Le système physique que nous exploitons avec BigDFT se compose de 144 orbitales. Le tableau 7.3 montre la répartition optimale de ces 144 orbitales entres les différentes tâches MPI pour avoir les meilleures performances. Cette répartition change en fonction de la puissance relative CPU/GPU. Ainsi, pour deux machines basées sur des processeurs de puissance différente, le calcul de répartition doit être entièrement refait.

Les résultats obtenus pour ces deux répartitions sont représentés dans les deux tableaux 7.4 et 7.5.

Le tableau 7.5 propose de montrer le temps passé dans APPLY_PROJ en distinguant les tâches CPU et les tâches GPU. Ainsi, les cases du type “8.6/31” signifient que APPLY_PROJ a duré 8.6s sur les tâches GPU et 31s sur les tâches CPU.

Observation :

Nous constatons que dans le cas d'une répartition non homogène, lorsque l'on ajoute plus d'orbitales à une tâche GPU, les parties CPU deviennent plus lentes car elles traitent plus d'orbitales. Ainsi, le temps passé dans APPLY_PROJ dans une tâche GPU est sensiblement plus important que sur les tâches uniquement CPU. Cette observation nous sera utile pour comparer l'approche 1 à l'approche 2.

7.2.5 Comparaison des deux approches et analyse

Nous allons maintenant comparer les deux approches du point de vue des performances et de la facilité de mise en place. Celle qui offre le plus d'avantages sera intégrée dans BigDFT.

Performance Avec l'aide des tableaux 7.4 et 7.1.a, nous constatons que plus le nombre de CPU utilisé est grand devant le nombre de GPU, plus les performances de la version sans virtualisation (tableau 7.4, avec une répartition non homogène des données) se rapprochent de la version où la virtualisation de GPU est activée (tableau 7.1.a, avec une répartition homogène des données). Ainsi, pour 4 tâches MPI et 1 GPU, la version avec virtualisation

Xeon X5570 (Nehalem)		
	1 GPU	2 GPU
Tâches MPI CPU + GPU	46 orbitales	51 orbitales
Tâches MPI CPU	14 orbitales	7 orbitales
Xeon L5420 (Harpertown)		
	1 GPU	2 GPU
Tâches MPI CPU + GPU	60 orbitales	54 orbitales
Tâches MPI CPU	12 orbitales	6 orbitales

Table 7.3 – Répartition optimale des orbitales pour des machines basées sur des Xeon X5570 (Nehalem) et des Xeon L5420 (Harpertown).

Processeur	**Xeon X5570 (Nehalem)**			**Xeon L5420 (Harpertown)**		
Nb. tâches MPI	1	4	8	1	4	8
TOTAL						
0 GPU (s)	573.00	160.00	91.00	892.00	272.00	214.00
1 GPU (s)	x	102	82	x	142	103
2 GPU (s)	x	70	68	x	86	81

Table 7.4 – Temps d'exécution de TOTAL pour une répartition non homogène des données. La virtualisation n'est pas utilisée ici.

Nb. tâches MPI	1	4	8
APPLY_PROJ CPU / GPU			
0 GPU	71,00 /x	19.25 / x	11.88 / x
1 GPU	x	9.0 / 46.7	8.6 / 31
2 GPU	x	7.7 / 29.7	5.0 / 26.0

Table 7.5 – Temps d'exécution de APPLY_PROJ pour les tâches CPU et GPU pour une répartition non-homogène avec des processeurs Xeon L5420 (Harpertown).

dure 94s alors que la version sans virtualisation dure 102s. Au contraire, pour 8 tâches et 1 GPU, la version avec virtualisation a besoin de 90s pour finir alors que l'autre ne met que 82s.

Comme vu plus haut, lorsque nous avons peu de tâches MPI, les parties CPU des tâches GPU durent beaucoup plus longtemps car ces tâches ont plus d'orbitales. Il y a donc beaucoup d'attente due à la synchronisation MPI entre les tâches. Lorsque l'on augmente le nombre de tâches MPI, le temps d'exécution CPU diminue car il est parallélisé entre toutes les tâches. Ainsi, les temps d'attente dus aux synchronisations MPI deviennent beaucoup moins coûteux et le temps total diminue.

Les performances des deux approches sont donc largement comparables : la virtualisation a un avantage lorsque il y a peu de tâches MPI par GPU. Pour le cas inverse, c'est la version sans virtualisation qui est légèrement devant. Ainsi, le choix entre l'une et l'autre doit être fait en fonction de la facilité de leurs intégrations dans les applications.

Facilité de mise en place Sans virtualisation d'accélérateurs, la répartition des orbitales entre chaque tâche MPI doit être précisément calculée afin d'assurer un équilibrage de la charge entre les tâches accélérées et les tâches non accélérées. Cette répartition dépend du nombre de tâches MPI créées et du ratio de puissance entre CPU et GPU. Ce calcul de répartition est très fastidieux et n'est pas généralisable à tous les codes : un code aussi flexible que celui de BigDFT n'est pas commun.

Les applications bâties sur le modèle MPI s'attendent pour la plupart à avoir des tâches MPI de puissances homogènes. Nous avons vu qu'un équilibrage statique entre des tâches MPI est difficile. Un équilibrage de charge dynamique entre les tâches MPI peut être envisagé mais n'est pas prévu dans le modèle MPI. Cela nécessite des adaptations profondes du code des applications ou l'utilisation de supports exécutifs particuliers comme AMPI [CH03]. En donnant l'illusion à chaque tâche d'avoir les mêmes ressources de calcul, la virtualisation de GPU proposée par S_GPU 1 permet d'exposer une architecture virtuelle bien adaptée aux applications basées sur le modèle de programmation MPI, et de permettre à celles-ci d'utiliser les outils standards MPI.

Sur un nœud hybride, l'approche homogène avec la virtualisation réalisée par S_GPU 1 permet d'atteindre de très bonnes performances et une répartition homogène peut être conservée. Ainsi, nous avons donc choisi d'utiliser cette approche dans la version de BigDFT destinée à être utilisée en production. Par conséquent, la version hybride de BigDFT utilise la virtualisation de GPU associée à une répartition homogène des données.

Dès lors, nous allons nous baser sur la version de BigDFT qui exploite la virtualisation d'accélérateur pour évaluer son comportement sur une grappe hybride.

Il faut noter que nous parlons ici d'un code de production qui est utilisé pour simuler des objets utiles aux physiciens.

7.3 Évaluation de BigDFT sur des grappes hybrides

Nous avons utilisé le code hybride de BigDFT associé à S_GPU 1 pour nos expériences sur Titane. Notre objectif est de comprendre quel est le niveau de l'efficacité parallèle

d'un code hybride comme BigDFT, tant du point de vue des performances de calcul que du point de vue de la consommation énergétique. Pour analyser le comportement de BigDFT, nous allons utiliser le mécanisme de traçage mis en place dans S_GPU 1. Nous allons, tout d'abord, exposer les performances atteintes sur une grappe hybride, puis nous montrerons, dans un second temps, comment observer l'exécution hybride de BigDFT. Une fois les performances en calcul évaluées, nous finirons par une analyse de la consommation énergétique d'une exécution hybride de BigDFT.

7.3.1 Contexte expérimental

Nous avons utilisé la grappe de calcul hybride TITANE, qui comporte, dans sa partie hybride, 96 nœuds connectés chacun à deux GPU NVIDIA d'un "cabinet tesla" S1070. Chaque nœud contient deux processeurs Intel Nehalem quadri-cœurs cadencés à 2.93 Ghz ainsi que 24 Go de RAM. Les nœuds de calcul sont interconnectés par un réseau InfiniBand DDR.

7.3.2 Performances atteintes par BigDFT sur TITANE

Nous proposons dans cette section de montrer le niveau de performance atteint par BigDFT sur une grappe hybride. Puis, nous analyserons l'accélération atteinte par la version hybride du code.

L'évaluation a été effectuée sur la grappe hybride Titane avec un jeu de données pour BigDFT comportant 576 orbitales. L'expérience consiste à garder le jeu de données constant tout en faisant varier le nombre de nœuds utilisés. Appelons n ce nombre de nœuds. Nous utilisons, avec S_GPU 1, 8 CPU par nœud ainsi que 2 GPU. Ainsi, pour un nombre n de nœuds utilisés, BigDFT tirera partie de $8.n$ processeurs et $2.n$ GPU.

Sur les nœuds de la grappe, nous utilisons deux GPU pour 8 cœurs CPU ; la bibliothèque S_GPU 1 va automatiquement attribuer à quatre processus le premier GPU et aux quatre autres le second GPU, et ce, sur chacun des nœuds.

Description des expériences Les quatre courbes de la figure 7.3 représentent toutes une accélération. La référence (accélération de 1) correspond au temps d'exécution de BigDFT lorsque 56 tâches MPI, réparties entre 7 nœuds hybrides, sont utilisées. Chaque tâche MPI utilise 1 cœur CPU. Notre évaluation est à volume de données constant : l'accélération idéale CPU a donc été calculée en considérant que lorsque les ressources de calcul sont augmentées d'un facteur x, le temps d'exécution est réduit de ce même facteur x.

Nous avons également calculé l'accélération idéale d'une exécution hybride de BigDFT. La référence (accélération de 1) est toujours le temps d'exécution de la version CPU de BigDFT utilisant 56 tâches MPI. Cependant, lorsque BigDFT exploite les accélérateurs GPU avec 56 tâches MPI, son temps d'exécution est 2,03 fois inférieur à la version CPU. Ainsi, le premier point de l'accélération idéale GPU est situé à un niveau d'accélération de 2,03. Les autres points sont calculés de la même manière que la version CPU. Nous constatons que la pente de l'accélération idéale GPU est exactement 2,03 fois celle de l'accélération idéale CPU.

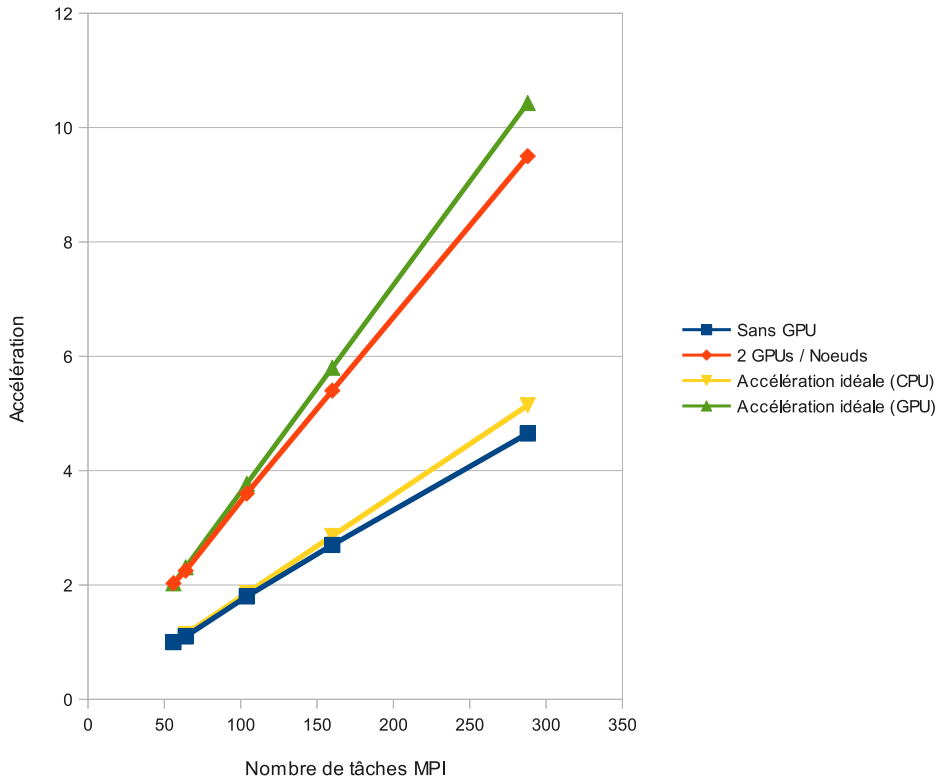


Figure 7.3 – Accélération de BigDFT sur Titane

Les deux autres courbes, qui correspondent à l'accélération mesurée CPU et GPU ont toutes pour référence (accélération 1) une exécution uniquement CPU de BigDFT sur 56 tâches MPI. Les points de mesure sont répartis entre 56 tâches MPI (7 nœuds, 14 GPU) et 288 tâches MPI (36 nœuds, 72 GPU). Chaque exécution CPU utilise tous les cœurs des nœuds (une tâche MPI par nœud). Toutes les exécutions hybrides utilisent tous les cœurs CPU et tous les GPU des nœuds.

Efficacité de la version CPU Nos expériences confirment la très bonne efficacité parallèle de BigDFT lorsque uniquement les CPU sont utilisés. En effet, l'accélération atteinte par BigDFT (courbe bleue) et l'efficacité idéale CPU (courbe jaune) sont très proches.

Efficacité de la version GPU La version GPU de BigDFT exhibe également une bonne efficacité : l'accélération atteinte par la version GPU de BigDFT (courbe rouge) et l'efficacité idéale GPU (courbe verte) sont très proches. Pour chaque point de mesure, la version hybride est environ deux fois plus rapide que la version CPU.

L'analyse détaillée du code de BigDFT montre que les parties portées sur GPU représentent 60 % du temps d'exécution total de BigDFT avec le système physique utilisé. Ainsi, l'accélération maximale que l'on peut atteindre pour BigDFT est d'un facteur de 2,5 (100/40), si l'on considère que l'accélération apportée par les GPU est telle qu'elle rend négligeable le temps d'exécution GPU par rapport au code CPU correspondant. L'accélération observée est d'un facteur 2, très proche du facteur 2.5 maximum calculé. Cela montre le bon comportement de la version accélérée du code de BigDFT.

Sur une machine qui possède 8 cœurs CPU et deux GPU, les bonnes performances de la version GPU des convolutions (cf. figure 6.3, page 107) permettent donc d'atteindre une accélération d'un facteur 2 sur le code complet de BigDFT.

La figure 7.3 montre que l'efficacité parallèle de la version accélérée reste similaire à la version non accélérée. Ainsi, même si le poids des communications MPI reste identique pour les deux versions de BigDFT et que le temps des calculs a diminué, cela n'influence pas l'efficacité. Dans d'autres codes, où le poids des communications est beaucoup plus important que dans BigDFT, l'efficacité parallèle d'une version accélérée pourrait diminuer par rapport à une version CPU.

Les particularités des accélérateurs, comme les GPU, nécessitent des outils spécifiques pour s'assurer qu'une application les exploite correctement. Un des problèmes majeurs de performance est lié aux temps de transferts coûteux entre les mémoires CPU et GPU. Une des solutions pour la résolution de ce problème est le recouvrement calcul-transfert ; des outils adaptés sont nécessaires pour observer ce recouvrement.

7.3.3 Visualisation des exécutions

Analyser une exécution en la visualisant permet de détecter des problèmes mais également de confirmer qu'une exécution se comporte comme prévu. Nous allons, dans cette partie, montrer comment nous avons employé des outils de visualisation pour analyser des exécutions de BigDFT. Différents outils sont disponibles et adaptés à différents types d'analyse. Ainsi, nous avons utilisé deux outils de visualisation lors de nos expériences. Le

premier est ITAC (Intel Trace Analyser and collector), qui est très efficace pour l'étude de programme MPI. Le second est l'outil intégré à S_GPU 1, qui a été spécialement réalisé pour analyser des applications S_GPU 1.

7.3.3.1 Visualisation d'une exécution sans virtualisation

Dans la section 7.2.4, nous avons décrit une expérience dans laquelle la virtualisation n'était pas utilisée : il fallait modifier la répartition des données attribuées aux tâches MPI pour obtenir les meilleures performances.

Pour interpréter les résultats, le temps d'exécution des différentes tâches MPI et le surcoût dû aux synchronisations sont des indices intéressants. Pour obtenir ces informations, nous avons utilisé l'outil ITAC.

La figure 7.4 est une capture d'écran d'un graphique produit par ITAC. Elle représente deux exécutions de BigDFT, pour lesquelles une tâche MPI utilise un GPU et les sept autres utilisent uniquement les CPU. Dans la fenêtre du haut, BigDFT a été configuré afin que toutes les tâches MPI traitent le même nombre d'orbitales. Dans la fenêtre du bas, les orbitales ne sont pas réparties de manière homogène entre les tâches MPI : la tâche exploitant le GPU en possède plus.

Répartition homogène des données Dans la fenêtre supérieure c'est la tâche MPI numéro 6 qui utilise le GPU. Nous observons que les parties accélérées, comme le préconditionneur ("Precondition" sur le graphique), sont extrêmement rapides sur la tâche 6 (il dure 2 s). Cependant, les autres tâches n'étant pas accélérées, les temps de synchronisation MPI sont importants. En effet, après le préconditionneur, une barrière de synchronisation globale impose à la tâche accélérée d'attendre les autres tâches : cela annule donc l'intérêt des GPU (le préconditionneur dure 14 s dans ce type de tâches).

Répartition non-homogène des données Nous étudions maintenant la fenêtre du bas, qui représente une exécution de BigDFT dans laquelle nous avons réparti les orbitales de manière optimale. C'est la tâche MPI 3 qui utilise le GPU. Nous sommes sur TITANE, et les processeurs sont de type Nehalem : comme décrite dans le tableau 7.3, la répartition optimale pour cette configuration consiste à attribuer 46 orbitales à la tâche 3 et 16 orbitales aux autres. Puisque la tâche GPU traite plus d'orbitales, cette répartition augmente le temps de ApplyProj (qui, comme nous l'avons vu, n'est pas accéléré). Les autres tâches traitant moins d'orbitales, leurs temps d'exécution sont faibles.

Nous observons que la répartition optimale n'annule pas les temps d'attente MPI, elle ne fait que les diminuer. En effet, en augmentant le nombre d'orbitales, le temps de synchronisation engendré par le préconditionneur va diminuer, mais celui engendré par ApplyProj va considérablement augmenter. A l'inverse, réduire le nombre d'orbitales affectées à la tâche GPU va diminuer le temps de synchronisation dû à ApplyProj, mais augmenter celui du préconditionneur.

La distribution optimale d'orbitales est obtenue lorsque le temps des synchronisations MPI est minimum. Un outil de visualisation adapté, comme ITAC, est essentiel pour déterminer cette répartition. Cependant, ITAC ne permet pas la visualisation de l'activité

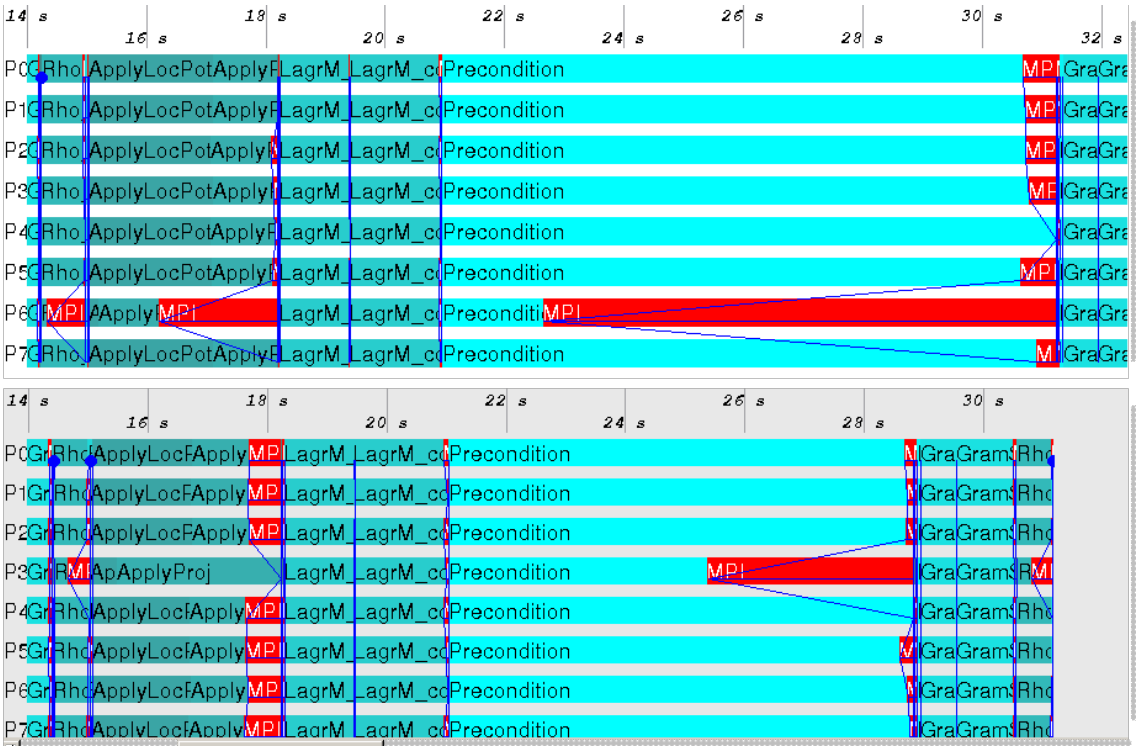


Figure 7.4 – Deux exécutions de BigDFT sans la virtualisation d'accélérateurs. Capture d'écran d'ITAC.

des GPU. Pour avoir une vue globale de l'application, ITAC suffit largement, pour étudier en détail l'utilisation des GPU, d'autres outils sont nécessaires.

7.3.3.2 Visualisation d'une exécution avec la virtualisation S_GPU 1

Notre objectif est de vérifier de manière expérimentale que des recouvrements calcul-transfert ont lieu. Il s'agira aussi de montrer que l'exécution de BigDFT est bien équilibrée sur notre cluster hybride, et qu'ainsi, notre approche basée sur la virtualisation permet bien d'équilibrer la charge entre les différentes tâches MPI. Nous verrons enfin comment l'outil de visualisation permet de mettre en évidence le déroulement des opérations CPU et GPU d'une exécution hybride.

La figure 7.5 représente une trace, produite par S_GPU 1 et interprétée par le logiciel ViTE. Sur cette trace, chaque nœud a deux GPU physiques. Chaque GPU physique est partagé, grâce à S_GPU 1, sous la forme de 4 GPU virtuels. La trace montre une exécution de BigDFT lancée sur quatre nœuds de la grappe hybride (32 tâches MPI se partagent 8 GPU). Nous voyons, pour chaque tâche, le travail des GPU virtuels et des CPU.

Exécution homogène La trace 7.5 permet d'analyser globalement le déroulement de l'application dans l'ensemble des nœuds utilisés de la grappe hybride. Les temps d'exécution des différentes parties du code de BigDFT sont identiques pour toutes les tâches MPI. Ainsi, APPLY_PROJ, qui utilise uniquement les CPU, traite le même nombre d'orbitales

pour chaque tâche et son temps d'exécution est donc similaire pour toutes les tâches. Nous pouvons faire la même observation pour PRECOND, qui lui n'utilise que les GPU. Ces observations confirment que notre approche basée sur une virtualisation de GPU permet à BigDFT de rester dans un mode où les tâches font toutes exactement les mêmes opérations aux mêmes moments. Chaque tâche MPI de BigDFT utilise donc la même puissance de calcul et l'exécution de BigDFT est bien équilibrée.

Recouvrements calculs transferts Pour étudier des zones particulières, ViTE permet d'agrandir des parties spécifiques de la visualisation et offre ainsi la possibilité d'étudier des points particuliers de l'exécution. Nous agrandissons fortement une zone et nous observons sur la figure 7.6 un recouvrement entre un calcul GPU effectué par la tâche MPI 15 et un transfert mémoire entre CPU et GPU effectué par la tâche MPI 31.

Il est ainsi possible, avec une application S_GPU 1, d'analyser son comportement, globalement au niveau de la grappe ou localement pour vérifier un point particulier de son exécution.

Exécution hybride CPU - GPU Sur la figure 7.5, les éléments "Exécution de PRECOND" correspondent au temps d'exécution total du calcul GPU de PRECOND, qui est constitué de plusieurs kernels CUDA. Cependant, nous observons que ce temps d'exécution contient bien l'exécution des kernels CUDA mais également du temps d'attente (en noir sur la figure). Ce comportement est normal car S_GPU 1 sérialise l'accès aux GPU : si une tâche a l'accès au GPU, lorsque une autre tâche MPI prévoit de lancer un kernel, elle est mise en attente.

Le modèle S_GPU 1 est synchrone, lorsque un GPU virtuel travaille, le CPU associé est en attente. Ainsi, lorsque une tâche MPI a lancé un kernel CUDA ou attend qu'un GPU soit libre, les CPU sont inutilisés. C'est pour lever cette limitation que nous avons mené le travail sur S_GPU 2.

Ils existe d'autres outils de visualisation, comme Vampir [1], qui, contrairement à ITAC, permettent d'obtenir des informations concernant MPI et les GPU. L'approche de traçage implantée dans S_GPU 1, qui est très liée au modèle de programmation de S_GPU 1, permet en plus d'étudier le comportement de chaque accélérateur virtuel. Ainsi, il paraît intéressant de combiner ces différents outils. Vampir, ayant des fonctions poussées d'analyse de code MPI, peut être utilisé dans un premier temps. Dans un second temps, l'application peut être analysée avec les outils de S_GPU 1 pour affiner la compréhension de l'exécution du code.

L'analyse de trace d'exécution grâce à des outils de visualisation permet d'appréhender le comportement d'un programme durant son exécution. La compréhension d'une exécution est essentielle pour optimiser les performances d'une application.

Nous allons, dans la partie suivante, étudier un autre type d'analyse : il s'agira de mesurer la consommation énergétique d'une machine hybride. Ainsi, il nous sera possible

1. `http://www.vampir.eu/`

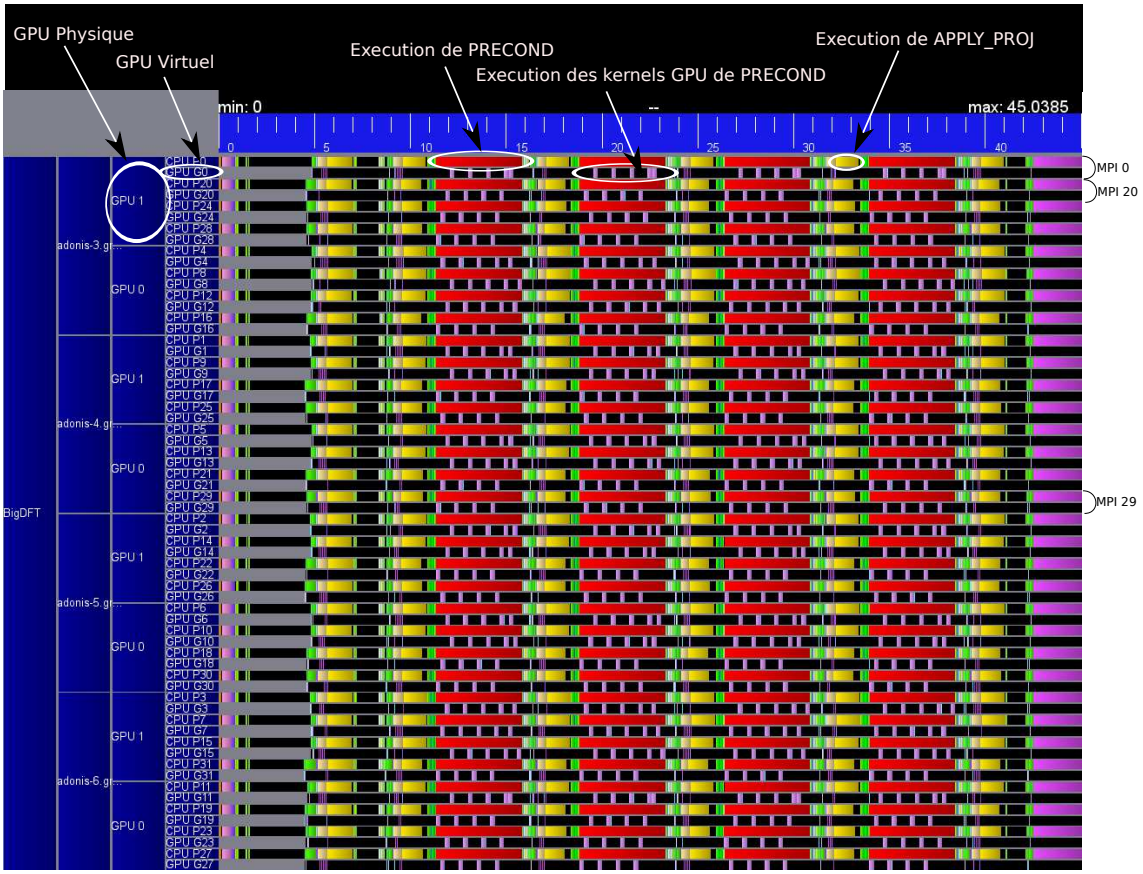


Figure 7.5 – Trace de BigDFT avec S_GPU 1 sur quatre nœuds d'un cluster hybride. La quantité de calcul est équilibrée ; chaque tâches MPI exécute les mêmes opérations.

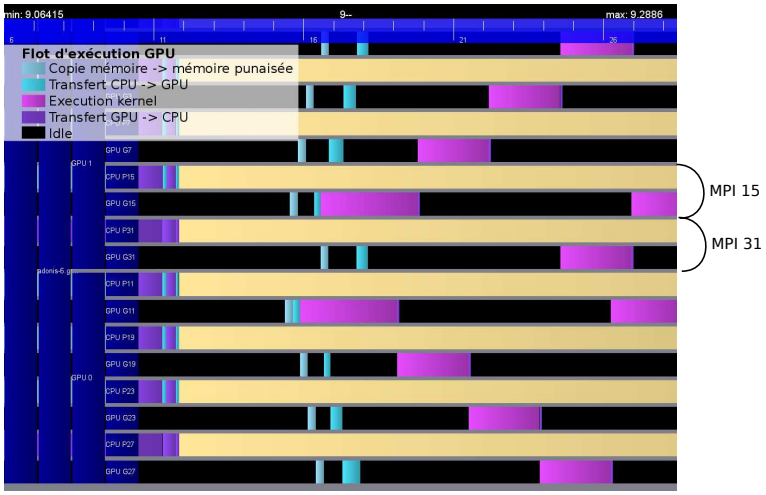


Figure 7.6 – Trace sur quatre nœuds d'un cluster hybride. Vue détaillée d'un recouvrement calcul - transfert.

d'observer les variations de la consommation énergétique d'une machine en fonction des phases de calcul de BigDFT.

7.3.4 Consommation énergétique d'une exécution de BigDFT

Dans cette section, nous allons analyser la consommation énergétique d'une machine sur laquelle est exécutée BigDFT. Nous allons mesurer globalement la consommation (CPU et GPU) d'une machine pour déterminer quel est l'effet de l'accélération apportée par l'utilisation des GPU sur la consommation énergétique.

7.3.4.1 Mesure de la consommation

Pour nos expériences, nous utilisons la machine basée sur le processeur L5420 (Harpertown) associée à un cabinet TESLA. Pour mesurer la consommation énergétique des GPU nous employons un dispositif permettant de mesurer la consommation d'appareils électriques. Notre appareil permet de déterminer l'intensité $i(t)$ en Ampère d'un courant passant dans des dispositifs électriques. L'appareil en question est un *APC Metered PDU*[2], sa précision de mesure est $0,1A$ soit $22W$. Nous connectons une ligne de mesure au cabinet TESLA, et une autre ligne à notre machine basée sur le processeur L5420.

Nous pouvons donc mesurer séparément :

— La consommation des quatre GPU du cabinet TESLA.

— La consommation de la machine basée sur le processeur L5420 (cela comprend les CPU, les disques durs, les dispositifs d'entrée/sortie, mais pas les GPU).

Il existe aujourd'hui dans les processeurs Intel des compteurs [Int12] permettant de récupérer des informations concernant la consommation énergétique. NVIDIA fournit également de telles possibilités, avec, par exemple, la bibliothèque NVML[3]. Ces facilités n'étaient pas disponibles lors de nos expérimentations, et nous avons donc dû utiliser un dispositif externe.

Pour avoir la puissance consommée en Watt, nous utilisons la formule suivante :

$$P(t) = i(t).U.cos(\phi) \tag{7.1}$$

Pour nos expériences, $U = 230V$ et $cos(\phi) = 0.98$. Pour une exécution de BigDFT de $t_{début}$ à t_{fin}, l'équation (7.2) donne l'énergie E consommée durant l'exécution en Joule.

$$E = \int_{t_{début}}^{t_{fin}} P(t)\,\mathrm{d}t \tag{7.2}$$

7.3.4.2 Consommation énergétique de BigDFT sur un nœud hybride

Nous avons mené deux expériences avec un jeu de données de BigDFT comportant 144 orbitales. La première expérience a consisté à exécuter la version hybride de BigDFT (CPU et GPU sont exploités pour le calcul). Pour la seconde expérience, seuls les CPU ont été utilisés pour le calcul.

2. APC - Metered Rack PDU, `www.apc.com/`

3. NVIDIA Management Library `http://developer.nvidia.com/cuda/nvidia-management-library-nvml`

Exécution hybride CPU - GPU Pour cette expérience, nous utilisons la version S_GPU 1 de BigDFT que nous exécutons sur huit cœurs CPU et deux GPU. Les résultats sont indiqués sur le graphique 7.7. Les deux courbes représentent la consommation du nœud et la consommation des GPU du cabinet TESLA. On peut y voir une consommation à vide de 220W pour le nœud et de 300W pour les quatre GPU du cabinet TESLA.

Le modèle d'exécution de S_GPU 1 étant synchrone, lorsque les GPU travaillent, les CPU se mettent en attente ; ce comportement est bien visible sur le graphique 7.7, lorsque les GPU consomment le plus (ils sont actifs), les CPU consomment le minimum et vice-versa. Nous utilisons l'équation (7.2) pour calculer l'énergie consommée par l'exécution représentée par la figure 7.7. En intégrant sur la durée d'exécution nous obtenons les résultats suivants :

Consommation énergétique pour 8 cœurs CPU et deux GPU :

Énergie GPU : 28kJ

Énergie CPU : 19kJ

Énergie totale pour l'exécution de BigDFT : **47kJ**

Exécution uniquement CPU Nous avons ensuite lancé une exécution de BigDFT en désactivant l'usage des GPU. Cependant, dans notre grappe hybride, il est impossible d'éteindre nos GPU, ceux-ci sont donc en attente et consomment une puissance de 300W.

Consommation pour 8 CPU, les GPU sont inutilisés :

Énergie GPU : 60kJ (non utilisés mais consomment de l'énergie)

Énergie CPU : 59kJ

Énergie totale pour l'exécution de BigDFT (avec les GPU inactifs) : **119kJ**

Énergie totale sans prendre en compte les GPU inactifs : **59kJ**

Lorsque les GPU de type 1070 sont sous tension mais inutilisés, la consommation d'une exécution est 2.5x plus importante qu'une exécution qui tire partie des GPU. Cela est évident mais un GPU inutilisé consomme énormément d'énergie au repos. Aujourd'hui, cette appréciation doit être nuancée car les nouvelles générations de GPU (comme les TESLA M2070 ou KEPLER) ont un mode veille qui diminue leurs consommations lorsqu'elles sont inutilisées.

Si l'on se place dans le cadre d'une machine sans GPU (on considère que la consommation des GPU est 0J), on arrive à une consommation de 59kJ, qui est là encore supérieure aux 47kJ consommés par notre code hybride.

Ainsi, dans le cas de BigDFT, les GPU aident à consommer globalement moins d'énergie. Cette consommation d'énergie est directement liée au temps de la simulation (puisque nous intégrons sur le temps d'après l'équation (7.2)). Ainsi, la version GPU de BigDFT étant plus rapide que la version CPU, elle consomme moins d'énergie malgré la puissance électrique requise par les GPU. Il faut bien noter cependant que la puissance maximale à fournir pour l'exécution d'un code GPU est de 450W, supérieure aux 300W de la version CPU. Sur nos machines d'expérimentation, nous réduisons le temps d'exécution tout en diminuant la consommation énergétique globale d'une exécution de BigDFT. Ainsi, l'utilisation des GPU augmente l'efficacité énergétique de la plateforme : le nombre d'opérations flottantes par unité énergétique est augmentée grâce aux GPU.

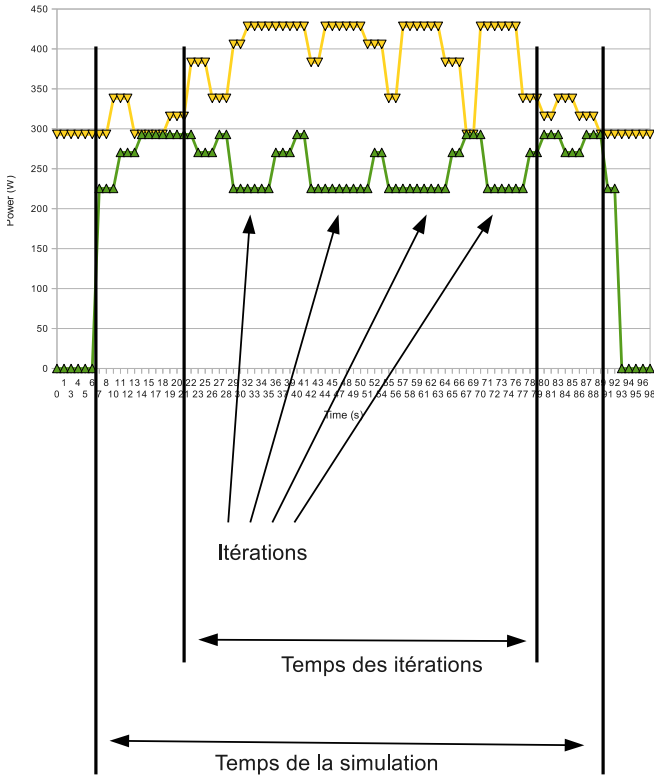


Figure 7.7 – Puissance consommée par une exécution de BigDFT lors de 4 itérations. On peut voir de manière séparée la consommation du nœud (CPU, mémoire, disque durs...) et la consommation des GPU.

Dans cette section, nous avons étudié l'influence des accélérateurs de type GPU sur la consommation énergétique. Les résultats sont clairs ; quand une bonne accélération est réalisée par un code utilisant les GPU – comme BigDFT – la consommation énergétique diminue par rapport à une utilisation purement CPU. Cependant, la puissance demandée au nœud est augmentée : lorsque 2 GPU travaillent simultanément, une puissance de près de 450W est requise. Cette puissance doit être prise en compte car certains systèmes de distribution électrique de grappes de calcul peuvent avoir plus de facilité à fournir une faible puissance longtemps plutôt qu'une forte puissance ponctuellement.

7.4 Conclusion

Les expériences menées dans ce chapitre avec l'application BigDFT montrent qu'il est tout à fait pertinent d'utiliser un modèle de programmation basé sur la virtualisation de GPU afin de conserver un modèle SPMD homogène sur des grappes de calcul hybrides. Distribuer les données d'une manière hétérogène est relativement complexe et ne montre pas de gains de puissance substantiels par rapport à l'approche homogène. Avec la virtualisation, une application peut donc conserver une répartition de données prévue pour une architecture homogène ce qui réduit considérablement l'intrusivité du passage vers une architecture hybride.

Ainsi, la voie choisie pour BigDFT a été celle de la virtualisation. Par conséquent, la version de BigDFT avec S_GPU 1 a été la version déployée dans les grandes grappes de calcul hybrides. Il faut noter que cette version est encore utilisée en production aujourd'hui dans certains centres de calcul.

L'application BigDFT a depuis évolué : les convolutions ont été portées en OpenCL, ce qui permet à BigDFT d'utiliser des GPU NVIDIA, ATI et toutes autre architecture supportant OpenCL. La plupart des convolutions peut également utiliser OpenMP, ce qui autorise une tâche MPI à utiliser plusieurs cœurs CPU. Ces différentes évolutions sont importantes pour espérer exploiter les prochaines générations de processeurs qui deviendront très probablement many-core.

Dans ce chapitre, nous avons également constaté que l'utilisation d'accélérateurs de type GPU permet de réduire la consommation énergétique nécessaire à une exécution de BigDFT, et plus généralement, à toute application utilisant les GPU d'une manière efficace. Depuis les expériences que nous avons menées, NVIDIA a fait un effort particulier pour réduire la consommation de ses GPU au repos. Par exemple, sur les GPU de la génération FERMI, comme les NVIDIA M2090, la carte peut automatiquement réduire sa fréquence de fonctionnement lorsqu'elle est inutilisée et sa consommation peut tomber jusqu'à 30W. Ces améliorations renforcent encore l'intérêt des GPU dans la diminution de la consommation énergétique des centres de calcul.

Nous avons mesuré ici la consommation après une exécution. Il pourrait être intéressant de développer des modèles prédictifs permettant d'estimer une consommation énergétique avant le démarrage de l'application. Associée au gestionnaire de ressources de la grappe de calcul, cette information permettrait d'ordonnancer les démarrages des applications afin de maîtriser la consommation totale de la machine. Ces questions de consommation

énergétique sont cruciales aujourd'hui ; les machines exaflopiques nécessiteront des progrès considérables concernant la diminution de la consommation d'une grappe de calcul. Les architectures hybrides peuvent être une solution permettant de répondre à cette problématique.

Chapitre Huit

Évaluation de S_GPU 2 avec SPECFEM3D

Sommaire

Dans ce dernier chapitre, nous allons évaluer l'utilisation de S_GPU 2 [OKMD11] avec un second programme de simulation scientifique : le simulateur de propagation d'ondes sismiques SPECFEM3D, décrit au chapitre 6.

SPECFEM3D devra être adapté pour utiliser le modèle de programmation et l'interface de S_GPU 2. L'objectif du travail est de permettre à SPECFEM3D d'exploiter simultanément des unités de calcul GPU et CPU.

Aussi, nous commencerons par montrer les choix généraux réalisés concernant la parallélisation hybride de SPECFEM3D avec S_GPU 2. Il conviendra ensuite de préciser comment nous avons mis en place les calculs hybrides, basés sur des threads CPU et GPU. Ce qui nous amènera à évaluer les performances atteintes par cette version hybride de SPECFEM3D. Nous finirons par une présentation de l'intérêt du module de traçage de S_GPU 2 avec SPECFEM3D.

8.1 Principes de la parallélisation hybride de SPECFEM3D avec S_GPU 2

Nous présentons dans cette section les principes que nous avons suivis pour paralléliser de manière hybride SPECFEM3D. L'objectif de cette parallélisation est de maximiser les performances en ayant un temps de calcul CPU et un temps de calcul GPU proche pour chaque calcul hybride.

Il existe deux versions du code de SPECFEM3D : la première utilise uniquement les CPU, la seconde permet d'exploiter des GPU dans la boucle en temps de SPECFEM3D. Pour notre travail de parallélisation hybride de SPECFEM3D avec S_GPU 2, nous sommes partis de la version GPU du code, et plus précisément de celle qui a été adaptée à S_GPU 1. Cette version, décrite au chapitre 6, consiste en un code Fortran dont les noyaux de calcul GPU sont appelés avec S_GPU 1.

La section de SPECFEM3D demandant la plus grande puissance de calcul est la boucle en temps qui est constituée de trois parties (les détails de cette structure ont été abordés au chapitre 6, page 109). Comme nous l'avons vu, la deuxième partie de la boucle en temps s'occupe de calculer les vecteurs des forces élastiques dans chaque élément spectral du maillage. Ces vecteurs de force sont ensuite sommés pour chaque élément commun du maillage. Un système de coloriage permet de rendre cette somme parallélisable sur GPU. Cette partie concentre environ 90% du temps d'exécution de la boucle en temps [TKL08]. Les parties 1 et 3 sont responsables de la mise à jour des vecteurs globaux de la simulation.

Dans la version du code que nous avons choisie, ces trois parties calculent uniquement sur les GPU. L'idée de base a donc été de transformer chacune de ces trois parties afin qu'elles puissent tirer simultanément partie de CPU et de GPU. Cela a été réalisé en créant trois calculs hybrides S_GPU 2 : chacun de ces trois calculs hybrides utilise donc des threads CPU et des threads GPU.

Nous conservons ainsi la structure de la boucle de SPECFEM3D en définissant une succession de trois calculs hybrides : chaque calcul correspond à une des trois parties de la boucle. Les adaptations réalisées sur la boucle en temps sont représentées sur la figure 8.2, figure qu'il faut mettre en relation avec la boucle originale figure 8.1. Nous mettons en évidence sur la figure 8.2 la succession de trois calculs exploitant simultanément des ressources CPU et GPU.

Après avoir décrit la plateforme expérimentale que nous avons employée, nous présentons les deux étapes nécessaires à la réalisation de chacun des calculs hybrides S_GPU 2 dans SPECFEM3D, à savoir, la création des threads CPU et GPU et la mise en place de la cohérence mémoire entre CPU et GPU.

8.1.1 Contexte expérimental

La machine que nous avons employée pour mener nos expériences est basée sur deux processeurs INTEL Xeon X5660 (6 cœurs par processeur) avec 24 Go de mémoire. Nous avons utilisé jusqu'à quatre de ces machines, chacune étant connectée, via PCI Express, à deux GPU d'un "cabinet tesla" S1070 contenant quatre GPU. Ces GPU ont chacun 4 Go de mémoire et 240 cœurs. Les nœuds de calcul sont interconnectés par un réseau InfiniBand DDR. C'est sur ce calculateur que nous avons développé et évalué SPEC-

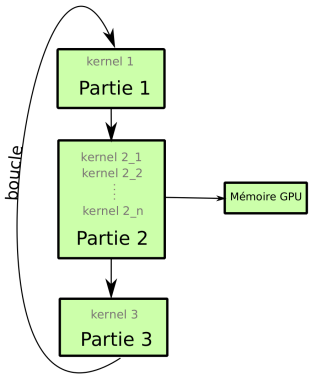


Figure 8.1 – Boucle principale de calcul exécutée uniquement sur un GPU

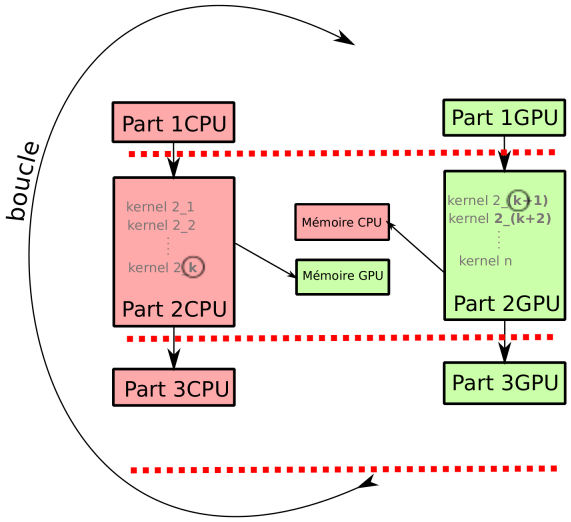


Figure 8.2 – Parallélisation hybride : les CPU et les GPU sont utilisés et des transferts de données supplémentaires doivent être ajoutés pour assurer la cohérence mémoire CPU et GPU. Après chaque ligne rouge pointillée, les deux zones mémoire doivent être cohérentes.

FEM3D/S_GPU 2.

8.1.2 Mise en place des threads CPU et GPU

La version de SPECFEM3D utilisée n'exploite que les GPU lors de phases intensives de calcul. Pour paralléliser de façon hybride SPECFEM3D avec S_GPU 2 nous avons créé des threads CPU à partir de la version CPU du code et des threads GPU à partir de sa version GPU. Ces deux types de threads, définis ci-dessous, sont contrôlés par S_GPU 2.

La mise en œuvre du modèle de programmation S_GPU 2 demande au programmeur de fournir des noyaux de calcul exécutables sur CPU et des noyaux exécutables sur GPU. La figure 8.3 montre globalement l'approche : des threads CPU et des threads GPU sont lancés par le programmeur. Ces threads sont exécutés simultanément sur chacune des ressources matérielles avec S_GPU 2.

Threads CPU Le modèle de programmation de S_GPU 2 étant basé sur une programmation hybride MPI et OpenMP, nous avons exploré une approche basée sur la coopération de ces deux environnements ; la figure 8.3 illustre un exemple où trois threads CPU sont lancés. Nous avons donc parallélisé avec OpenMP les parties 1, 2 et 3 du code CPU de SPECFEM3D qui étaient initialement séquentielles. Cela a été réalisé grâce à l'ajout de quelques directives OpenMP dans le code Fortran de SPECFEM3D. Des précisions sur la parallélisation OpenMP et une évaluation de l'efficacité parallèle du code seront apportées dans la deuxième partie de ce chapitre.

Threads GPU Plusieurs possibilités sont envisageables pour exploiter un système multi-GPU avec une application MPI comme SPECFEM3D.

La première est de créer plusieurs threads GPU par tâche MPI. Ces threads seront exécutés sur les GPU du nœud de calcul. Dans ce cas, le nombre de tâches MPI par nœud peut ne pas être lié au nombre de GPU par nœud. Chaque tâche crée en effet des threads GPU qui seront ordonnancés sur des GPU libres.

Une autre approche, qui est représentée sur la figure 8.3, consiste à créer un thread GPU par tâche MPI et de créer au moins autant de tâches que de GPU dans le nœud. Cette approche est adaptée dans le cas de code régulier où la charge de calcul GPU est équilibrée entre les tâches MPI. SPECFEM3D étant un code équilibré, c'est cette approche que nous avons utilisée dans nos expérimentations : chaque tâche MPI de SPECFEM3D crée au plus un seul thread GPU.

Calcul hybride Contrairement au code original de la figure 8.1, la version hybride de SPECFEM3D (figure 8.2) est constituée, pour chacune des trois parties de la boucle en temps, de threads CPU et GPU. Une exécution du code de SPECFEM3D adaptée à S_GPU 2 consiste donc en plusieurs tâches MPI par nœud, chacune d'entre elles lançant plusieurs threads CPU (OpenMP) et un thread GPU. Les threads CPU et GPU sont gérés par S_GPU 2.

Notre objectif est donc de réduire le temps d'exécution de la version GPU de SPECFEM3D en profitant de la puissance de calcul de nos CPU.

Les trois parties de la boucle en temps sont exécutées simultanément sur CPU et GPU et vont chacune traiter simultanément des données situées dans des espaces mémoire disjoints. Un protocole de cohérence mémoire doit dont être mis en place pour s'assurer que les calculs traitent des données correctes.

8.1.3 Cohérence mémoire

Dans l'algorithme de calcul d'éléments finis de SPECFEM3D, la phase de calcul des vecteurs de forces élastiques (partie 2) nécessite des tableaux d'entrée qui ont été entièrement mis à jour. De même, la phase de mise à jour des tableaux (parties 1 et 3) nécessite des tableaux d'entrées dans lesquels toutes les forces calculées sont enregistrées. Ainsi, après chacune des trois parties de la boucle en temps, les zones mémoire CPU et GPU doivent être cohérentes, c'est-à-dire, dans notre cas, identiques. Cette propriété est représentée par des lignes pointillées sur la figure 8.2. Des calculs différents peuvent donc s'effectuer sur les CPU et les GPU pour chacune des trois étapes de la boucle en temps. Néanmoins, à la fin de chacune des trois étapes, des opérations de communication entre les zones mémoire CPU et GPU ont lieu afin qu'elles soient identiques. De plus, une barrière de synchronisation est implantée à la fin de chaque étape pour assurer que les calculs CPU et GPU sont totalement terminés avant de passer à l'étape suivante. Par conséquent, cette approche se rapproche du modèle BSP [Val90].

Des opérations mémoire, représentées au centre de la figure 8.2 sont dès lors nécessaires pour assurer la cohérence entre mémoire CPU et GPU. Dans le modèle de programmation de S_GPU 2, la mise en place de ces opérations est à faire dans les fonctions de coupe et de regroupement de chaque calcul hybride. Ces deux fonctions ont été abordées à la section 4.2.3.1, page 63. Des précisions sur ces opérations mémoire seront amenées plus loin dans cette partie.

Nous avons vu, dans cette section, la méthodologie générale que nous avons suivie pour paralléliser de façon hybride SPECFEM3D avec S_GPU 2. Il faut en effet déterminer comment créer les différents types de threads et assurer la cohérence mémoire de l'ensemble. Cette méthodologie peut ainsi être employée pour des applications autres que SPECFEM3D. Après ces éléments généraux, nous allons montrer maintenant de manière plus précise comment SPECFEM3D a été adapté à S_GPU 2.

8.2 Éléments d'implémentation de SPECFEM3D/S_GPU 2

Nous allons dans cette section préciser les choix réalisés pour adapter SPECFEM3D au modèle de programmation de S_GPU 2, et ainsi permettre à cette application scientifique d'utiliser simultanément des CPU et des GPU.

Nous proposons de montrer comment les différentes parties de la boucle en temps ont été portées sur S_GPU 2. Pour chacune des trois parties, nous décrivons comment le calcul a été réparti entre CPU et GPU et quelles sont les opérations nécessaires à la cohérence mémoire de l'ensemble.

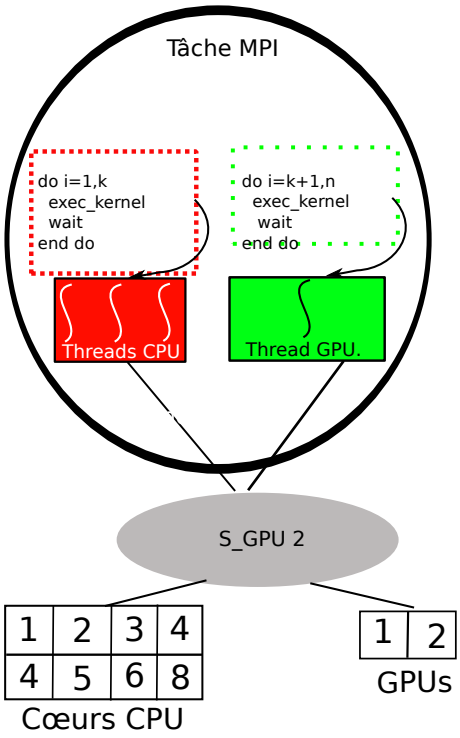


Figure 8.3 – Organisation des threads S_GPU 2 dans SPECFEM3D

8.2.1 Parallélisation OpenMP et création des threads GPU

Nous présentons tout d'abord comment les threads CPU et GPU sont organisés dans la version S_GPU 2 de SPECFEM3D. Ensuite, il s'agira de proposer une évaluation de la parallélisation OpenMP de notre implémentation.

8.2.1.1 Parallélisation hybride des parties 1 et 3 de la boucle en temps

Les parties 1 et 3 de la boucle en temps de SPECFEM3D sont assez semblables ; la mise à jour des vecteurs globaux consiste en une somme de deux tableaux (`veloc` et `displ`) dans lesquels chaque élément est multiplié par des coefficients constants. A l'initialisation de la boucle en temps, des transferts mémoire sont réalisés afin que les tableaux `veloc` et `displ` soient identiques sur CPU et GPU.

Threads CPU Comme nous l'avons vu, nous avons transformé le code séquentiel de SPECFEM3D de ces deux parties en code parallèle OpenMP. Une somme de tableaux séquentielle se transforme assez facilement en code parallèle avec OpenMP car chaque élément peut être traité en parallèle.

Le code parallélisé de la première phase de mise à jour des tableaux (partie 1 de la boucle en temps) est représenté ci-dessous :

```
1  !$OMP PARALLEL SHARED(displ,veloc,accel) PRIVATE(i,j)
2    do i=1,NDIM
```

```
3       !$OMP  DO
4       do j=1,NGLOB
5          displ(j,i) = displ(j,i) + deltat*veloc(j,i) + deltatsqover2*
               accel(j,i)
6          veloc(j,i) = veloc(j,i) + deltatover2*accel(j,i)
7          accel(j,i) = 0.
8       end do
9       !$OMP END  DO
10    end do
11 !$OMP END PARALLEL
```

Les threads CPU sont donc créés via les directives OpenMP.

Thread GPU Le thread GPU traitant la somme des tableaux contient exactement le code CUDA des parties 1 et 3 de la version GPU de SPECFEM3D.

Le calcul est ici répliqué : les threads CPU et le thread GPU réalisent les mêmes opérations. Les versions GPU et CPU des tableaux `veloc` et `displ` sont identiques avant cette phase, ils le sont donc également après. Comme cette phase représente uniquement 10% du temps de calcul, la réplication des calculs permet d'éviter des transferts de données qui auraient été nécessaires pour assurer la cohérence entre les zones mémoire.

8.2.1.2 Parallélisation hybride de la partie 2 de la boucle en temps

La partie 2 est l'étape la plus complexe de la boucle en temps : le calcul des vecteurs de force élastique prend en effet environ 90% du temps d'exécution de la boucle. Cette partie consiste en une succession d'appels à un kernel de calcul, chaque appel traitant les éléments du maillage d'une même couleur.

Dans la version S_GPU 2 de SPECFEM3D, chaque kernel va être exécuté soit sur des CPU soit sur des GPU. La figure 8.3 représente cette parallélisation : la tâche MPI lance une succession de kernels sur CPU et sur GPU. C'est à chaque invocation de kernels que les threads CPU et GPU sont créés.

Deux exécutions de kernels ne peuvent pas être simultanées lorsque ils sont exécutés dans le même espace mémoire. Ainsi, la fin de l'exécution d'un kernel doit être attendue avant d'exécuter un second kernel. C'est le sens du "wait" sur la figure 8.3. Comme dans notre cas, nous avons deux zones mémoire disjointes (CPU et GPU), cela autorise l'exécution simultanée, sans ajout de primitives de synchronisation, de kernels sur le CPU et d'autres sur le GPU.

La figure 8.3 permet de visualiser comment les kernels sont distribués entre CPU et GPU. Pour une partie 2 traitant n couleurs, n kernels doivent être exécutés. k kernels sont exécutés sur le CPU et $(n-k)$ kernels sur le GPU. (k étant un nombre entier qu'il convient de déterminer pour avoir un temps de calcul équilibré entre les CPU et les GPU).

Threads CPU Comme déjà abordé, chaque exécution de kernel GPU traite une couleur différente du maillage : cela permet une parallélisation poussée du kernel de calcul. Nous avons utilisé cette propriété pour paralléliser en OpenMP le code CPU du kernel de calcul de la partie 2. Chaque exécution du kernel de la partie 2 sur CPU est ainsi multithreadée.

```
1  !$OMP PARALLEL
2   do icolor = 1,k !nombre de couleurs a traiter sur CPU
3        nb_elem_color = num_elements_color(icolor)
4
5        !$OMP DO !code parallelise du kernel de calcul
6        do ispec = color_offset, color_offset + nb_elem_color -1
7
8        (...) code du kernel 2
9
10       end do
11 end do
12 !$$ END OMP PARALLEL
```

A la ligne 1, la variable k correspond au nombre de kernel à exécuter sur la partie CPU. Ensuite, le code du kernel commence à la ligne 6, et est parallélisé grâce à openMP. Ce code peut être mis en relation avec la figure 8.3 : les threads sont lancées dans la boucle, et le "wait" correspond à la barrière implicite définie par openMP après chaque boucle parallèle.

Thread GPU Le thread GPU contient le code CUDA du kernel de calcul de la version GPU de SPECFEM3D. Comme pour la version CPU, une boucle lance les threads GPU. $(n - (k + 1))$ threads sont créés.

```
1   do icolor = k+1,n !nombre de couleurs a traiter
2        call create_gpu_thread()
3        wait()
4
5  end do
```

Après l'exécution de la partie 2 telle que nous l'avons présentée, le résultat est distribué entre la mémoire CPU et GPU, c'est-à-dire que dans la mémoire centrale nous avons une partie des résultats et dans la mémoire GPU une autre partie. Il faut donc mettre en place des procédures pour que ces zones mémoire soient cohérentes. Ces procédures seront discutées en 8.2.2.

8.2.1.3 Efficacité de la parallélisation

Il s'agit maintenant de présenter globalement les performances de la parallélisation OpenMP des parties 1, 2 et 3 de la boucle en temps. Nous ne prenons pas ici en compte les temps nécessaires à la cohérence mémoire (cet aspect sera abordé en 8.3) mais seulement les temps de calcul. Nous mesurons le temps nécessaire pour effectuer 100 itérations de la boucle en temps : il y a donc une succession de 100 appels aux parties 1, 2 et 3 de la boucle. Nous faisons varier le nombre de threads OpenMP de 1 à 6 (les processeurs utilisés ont 6 cœurs). Les temps d'exécution sont représentés sur la figure 8.4. L'efficacité de notre parallélisation est donnée sur la figure 8.5.

Dans cette expérience, nous utilisons une tâche MPI. Les threads créés par cette tâche sont confinés dans une socket et accèdent à des données d'un seul banc mémoire NUMA. Nous évitons ainsi tout accès mémoire entre différents bancs NUMA.

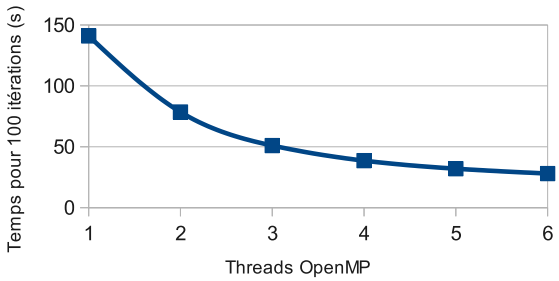


Figure 8.4 – Performance de la parallélisation OpenMP pour 100 itérations

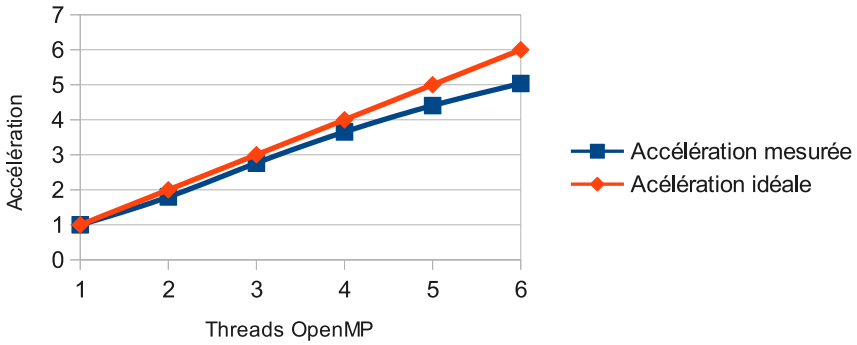


Figure 8.5 – Accélération du code OpenMP

– **Accélération parallèle** Nous observons sur la figure 8.5 que la pente de l'accélération diminue légèrement à partir de quatre threads. L'accélération reste cependant très proche de l'accélération idéale. Ainsi, d'après la figure 8.4, si quatre threads OpenMP sont exploités, SPECFEM3D sera capable de traiter 100 itérations en 38,5 s.

Dans les expériences qui seront présentées à la section 8.3 de ce chapitre, lorsque l'on double le nombre de tâches MPI, on double également la quantité de données à traiter. Ainsi, pour deux tâches MPI, en les positionnant chacune sur un banc NUMA distinct (mémoire et calcul), le temps d'exécution des deux tâches sera le même que le temps d'exécution montré sur la figure 8.4. Ceci est valable puisque les deux tâches MPI s'exécutent en parallèle sur deux bancs NUMA différents avec un volume de communication faible. Ainsi, une exécution de SPECFEM3D avec deux tâches MPI lançant chacune quatre threads traitera les 100 itérations de notre problème en 38,5 s.

Des mesures de performance montrent que lorsque l'on utilise deux tâches MPI par nœud et un GPU partagé entre ces deux tâches, les noyaux de calcul GPU de SPECFEM3D sont capables de traiter les 100 itérations en 12,9 s. Ainsi, dans cette configuration les performances CPU de SPECFEM3D ne sont pas négligeables par rapport aux performances GPU et utiliser les deux ressources de calcul semble pertinent.

Nous allons maintenant nous intéresser à la cohérence mémoire des calculs hybrides

S_GPU 2. Nous avons considéré deux stratégies permettant de réaliser cette cohérence mémoire qu'il convient maintenant de présenter. Les performances de ces stratégies seront analysées dans la troisième partie de ce chapitre.

8.2.2 Deux stratégies de cohérence mémoire

Dans notre mise en œuvre hybride de la boucle en temps, les résultats CPU et les résultats GPU doivent être cohérents. L'exécution des différentes parties de la boucle principale doit en effet pouvoir utiliser des données valides sur CPU et GPU. Cette section propose de présenter comment la cohérence mémoire a été mise en place pour les parties 1 et 3 avant de présenter deux approches de cohérence pour la partie 2 (appelées stratégie A et stratégie B).

Cohérence mémoire des parties 1 et 3 Plutôt que de découper le calcul en une partie CPU et une partie GPU puis d'effectuer des transferts pour assurer la cohérence des deux espaces mémoire, nous avons préféré répliquer le calcul et éviter ainsi de coûteux transferts. Les parties 1 et 3 sont donc exécutées entièrement sur CPU et sur GPU, et la cohérence mémoire est donc naturellement assurée.

Cohérence mémoire de la partie 2 L'organisation kernel/mémoire de la partie 2 est représentée sur la figure 8.6 : CPU et GPU traitent un ensemble différent de kernels. Cette figure permet de visualiser les kernels exécutés et les zones mémoire impliquées : la partie CPU traite les kernels 1 et 2 et la partie GPU le reste (kernel 3 à n). Dans chaque zone mémoire un tableau `accel` privé est défini, il y a donc, dans notre exemple, un tableau `accel` CPU et un tableau `accel` GPU. À la fin de l'exécution de tous les kernels de calcul, il faut que les différents espaces mémoire soient identiques.

Les données modifiées par la succession des exécutions du kernel de la partie 2 sont contenues dans le tableau `accel`. Ce sont donc les données de ce tableau qui doivent être identiques entre les mémoires CPU et GPU : les parties CPU et GPU contiennent en effet chacune un tableau `accel` local. À la fin de la partie 2, les deux tableaux `accel` locaux aux GPU et CPU doivent être identiques.

Deux actions sont nécessaires pour assurer cette cohérence :

1. Fusion des données. Un opérateur, permettant de fusionner les tableaux `accel` CPU et GPU, doit être fourni par le programmeur. La place de cet opérateur est la fonction de regroupement. Cet opérateur, qui est représenté sur la figure 8.6, doit prendre en compte le fait qu'un élément du tableau mémoire ait pu être modifié du côté CPU et du côté GPU. Sur la figure 8.6, un calcul CPU et un calcul GPU ont modifié tous deux la case mémoire 7 de leur version locale du tableau `accel`. Dans le tableau fusionné, la case numéro 7 doit donc réunir les résultats CPU et GPU.

2. Mise à jour des copies locales de `accel`. Les tableaux `accel` de la mémoire CPU et GPU doivent être mis à jour pour qu'ils soient identiques et contiennent les données fusionnées.

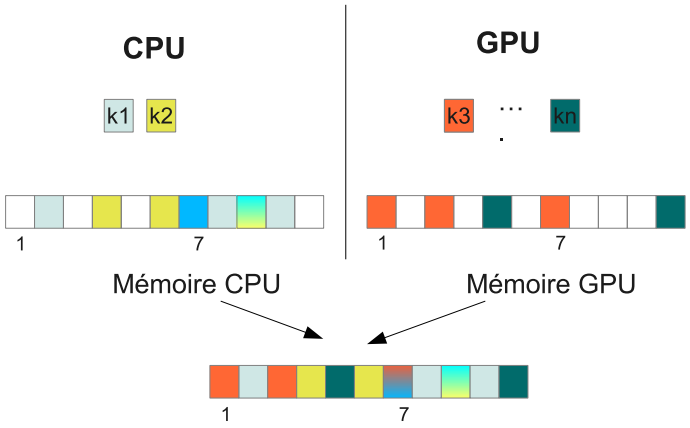


Figure 8.6 – Vue générale de l'hybridation de la partie 2 de SPECFEM3D

Pour assurer cette cohérence, et mettre en place la fusion des données et la mise à jour des tableaux, nous avons mis en place deux stratégies qui sont discutées dans la suite de cette partie.

8.2.2.1 Stratégie A : quelques « gros » transferts

Nous décrivons ici la première stratégie sur laquelle nous avons travaillé qui est représentée sur la figure 8.7. Les opérations effectuées par cette stratégie sont définies dans la fonction de regroupement du calcul hybride. Ainsi, les opérations permettant de fusionner les mémoires CPU et GPU sont effectuées après l'exécution des k kernels sur le CPU et des $(n-k)$ kernels sur le GPU.

Une propriété de l'algorithme de la partie 2 permet d'assurer que pour créer un vecteur $accel_{complet}$ qui comporte les contributions CPU (stockés dans $accel_{CPU}$) et les contributions GPU (dans $accel_{GPU}$), il suffit de sommer, éléments à éléments, ces deux vecteurs, ce qui peut s'écrire : $accel_{CPU}[1..n] + accel_{GPU}[1..n] = accel_{complet}[1..n]$.

Ainsi, pour mettre en œuvre la stratégie A, nous avons créé une fonction de regroupement exécutant les trois étapes suivantes (qui sont visibles sur la figure 8.7) :

1. Transfert de $accel_{GPU}$ depuis la mémoire GPU vers la mémoire CPU
2. Addition de $accel_{CPU}$ et $accel_{GPU}$ dans $accel_{complet}$
3. Transfert de $accel_{complet}$ depuis la mémoire CPU vers la mémoire GPU

Après l'étape 1 et 2, la mémoire centrale contient les contributions CPU et GPU dans $accel_{complet}$. Cependant, nous avons vu qu'à la fin de la partie 2, $accel_{GPU}$ et $accel_{CPU}$ doivent être identiques : c'est l'étape 3 qui réalise cela en transférant le tableau $accel_{complet}$ vers la mémoire GPU. Ainsi, puisque l'on transfère deux fois un tableau `accel` de n éléments, le volume de données transféré est égal à $2.n$.

Cette approche est simple à mettre en place mais elle implique des transferts de données que l'on peut qualifier d'inutiles. En effet, dans la mémoire CPU (respectivement dans la mémoire GPU), les kernels ne modifient qu'une partie du tableau `accel` (puisqu'ils sont répartis entre CPU et GPU). Ainsi, lors des étapes 1 et 3, des données non modifiées sont

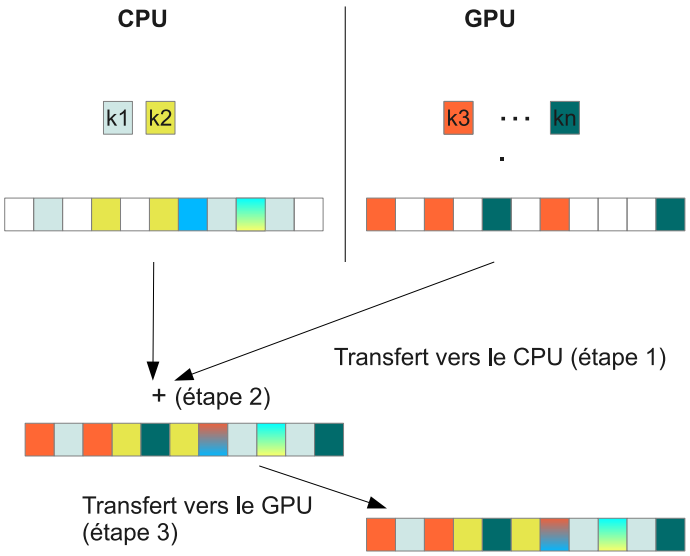


Figure 8.7 – Détails des mises à jour mémoire de la stratégie A

transférées.

Cette stratégie ne sélectionne pas de manière précise les données à transférer ce qui implique des transferts de zones mémoire non modifiées. Pour réduire ces transferts de données, nous avons mis en place une autre stratégie appelée stratégie B.

8.2.2.2 Stratégie B : plusieurs « petits » transferts

Nous présentons ici une deuxième stratégie qui a demandé des investigations plus poussées dans la compréhension du code de SPECFEM3D et qui est représentée sur la figure 8.8. L'objectif de cette stratégie est de réduire les transferts mémoire par rapport à la stratégie A et d'utiliser la capacité de recouvrement transfert-calcul des cartes GPU.

La cohérence mémoire est réalisée en trois étapes :

Étape 1 : calcul des éléments à transférer Cette étape consiste à déterminer, pour chaque kernel exécuté de la partie 2, un ensemble d'éléments du tableau `accel` modifiés par l'exécution d'un kernel particulier. Les éléments sont sélectionnés de telle sorte qu'ils ne seront pas modifiés par les kernels lancés par la suite. Il y a ainsi autant d'ensembles d'éléments que d'exécution de kernels.

Grâce à ces ensembles d'éléments, il est possible, lors de l'exécution d'un kernel 2, de transférer immédiatement les éléments de ces ensembles dans un autre espace mémoire. Ce transfert est valide puisque ces ensembles d'éléments garantissent que les données transférées ne seront pas modifiées dans la suite du calcul de la partie 2 de la boucle en temps.

Sur la figure 8.8, la première exécution du kernel (appelé k1), modifie les éléments (2,7,8,9,10) du tableau `accel`. Cependant, l'élément 9 est également modifié par l'exécution

suivante du kernel (appelé k2). Ainsi, seuls les éléments (2,7,8,10) sont transférés à la fin de k1. L'élément 9 est transféré ensuite, lorsque k2 a fini son exécution.

Le calcul des ensembles doit traiter plusieurs tableaux contenant plusieurs millions d'éléments. Elle est très coûteuse en temps et est donc lancée une fois à l'initialisation de SPECFEM3D, bien avant la boucle principale du code.

Étape 2 : transferts de données partiels Nous avons modifié le code du kernel 2 afin qu'après le calcul proprement dit, les CPU (respectivement les GPU) envoient les éléments du tableau `accel`, calculés à l'étape 1, vers la mémoire GPU (respectivement CPU). L'exécution du kernel suivant peut se faire pendant le transfert mémoire car, par construction, l'étape 1 crée un ensemble d'éléments qui ne seront pas modifiés par l'exécution d'un kernel suivant.

Les éléments à transférer sont répartis de manière non contiguë dans le tableau `accel`. Plutôt que d'initier un grand nombre de transferts d'un seul élément, peu efficace, nous avons préféré regrouper chaque élément à transférer dans un tableau intermédiaire avant d'initier le transfert de ce tableau intermédiaire. Nous considérons sur la figure 8.8, quatre exécutions de kernel (k1, k2, k3 et kn), il y a donc quatre transferts partiels de données contenus dans des tableaux intermédiaires.

La stratégie B consiste donc à lancer un kernel, puis, à la fin de son exécution, de transférer, grâce aux informations calculées à l'étape 1, les données qu'il a modifiées. Simultanément au transfert, le kernel suivant peut être lancé. Cela permet deux optimisations par rapport à la stratégie précédente :

1. Optimiser l'exécution grâce à des recouvrements calcul-transfert, puisque lors d'un transfert, l'exécution d'un kernel a lieu.
2. Réduire les données transférées : l'étape 1 construit une information qui permet de transférer uniquement des points modifiés. Ainsi, contrairement à la stratégie A, seul les éléments de `accel` indispensables sont transférés ce qui réduit le volume de données à transférer.

Étape 3 : Fonction de regroupement Les transferts mémoire étant fractionnés et lancés directement après chaque kernel, la fonction de regroupement n'initie aucun transfert. Elle a cependant la charge de regrouper ensemble chaque fraction de données, et ce sur la partie CPU et sur la partie GPU. Comme les données modifiées ne sont pas contiguës en mémoire, nous avons vu que les données étaient transférées grâce à des tableaux intermédiaires. La fonction de regroupement va donc reconstruire les données à partir de ces tableaux avant de les fusionner.

La stratégie B réduit la taille des transferts mais complexifie grandement son intégration dans SPECFEM3D. Plusieurs calculs supplémentaires, absents de la stratégie A, sont en effet nécessaires. L'évaluation de ces deux stratégies, dans la partie suivante, nous permettra ainsi de comprendre la meilleur voie à adopter.

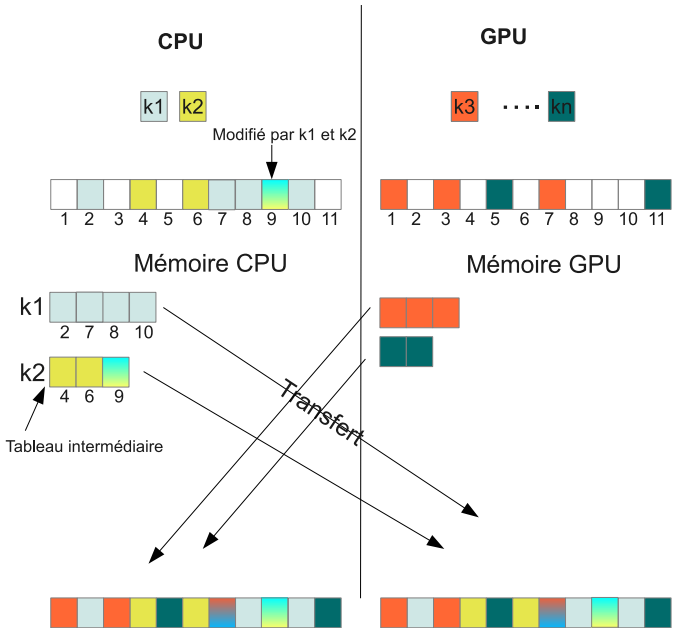


Figure 8.8 – Détails des mises à jour mémoire de la stratégie B

8.3 Performance de SPECFEM3D/S_GPU 2

Dans cette partie, nous allons évaluer les performances des calculs hybrides de notre version de SPECFEM3D adaptée à S_GPU 2. Il s'agira pour cela d'évaluer les stratégies A et B. L'objectif est de comprendre s'il est préférable d'avoir un code de cohérence mémoire simple, et donc de communiquer des données inutiles, comme la stratégie A, ou, au contraire, d'avoir un code de cohérence mémoire complexe pour communiquer uniquement ce qui a été modifié, comme la stratégie B.

Nous allons, pour cela, mener plusieurs expériences qui consistent à faire varier la quantité de calculs attribuée à la partie GPU et à la partie CPU. Après la description des expériences, nous présentons les performances atteintes par la stratégie A et la stratégie B en prenant soin d'analyser le poids des transferts de données et des opérations de mise à jour mémoire. La partie finale de cette étude est une comparaison des deux stratégies avec le code original de SPECFEM3D sur un calculateur hybride.

8.3.1 Description des expériences

Dans ces expériences, il s'agit de faire varier le nombre d'exécutions de kernel de la partie 2 attribué aux CPU et aux GPU. Dans le système physique utilisé, nous avons une succession de 25 kernels à lancer ; nous commençons donc, pour chaque tâche MPI, par exécuter un kernel sur les threads CPU et 24 sur le thread GPU, puis deux sur le CPU et 23 sur le GPU et ainsi de suite jusqu'à 24 sur le CPU et 1 sur le GPU. Pour évaluer l'effet de la puissance CPU nous utilisons tout d'abord 2 threads OpenMP par tâche MPI (ce qui simulera un CPU de faible puissance) puis 5 threads par tâche MPI (nous aurons ici un CPU de plus grande puissance).

Dans nos expériences, le matériel est basé sur une architecture comportant deux bancs NUMA. Comme déjà abordé en 8.2.1.3, nous créons deux tâches MPI : une tâche par banc NUMA. Les threads OpenMP créés par chaque tâche sont positionnés exclusivement sur les processeurs d'un même banc NUMA. Nous évitons ainsi tout effet d'accès distant entre deux bancs NUMA dans notre évaluation. Pour renforcer la puissance des CPU par rapport aux GPU, nous n'utilisons qu'un seul GPU qui sera partagé entre les deux tâches MPI. Les GPU sont en effet très performants et réduire le nombre de GPU par rapport au nombre de CPU nous permettra de renforcer l'effet des performances CPU de notre parallélisation hybride SPECFEM3D/S_GPU 2. De plus cette dissymétrie entre le nombre de GPU utilisé et le nombre de bancs NUMA est courant sur les grosses machines NUMA qui possèdent des dizaines de bancs NUMA et seulement quelques GPU.

8.3.2 Performances des deux stratégies de cohérence mémoire

Nos mesures de performance sont représentées sur la figure 8.9. Pour chaque graphique, l'axe des abscisses représente le nombre de kernels exécutés sur les cœurs CPU, il est donc gradué de 1 à 24. Pour connaître le nombre de kernels lancés sur le GPU, il suffit de retrancher chaque valeur au nombre 25.

Pour chaque répartition de calcul entre CPU et GPU, le temps nécessaire pour effectuer 100 pas de temps – donc 100 itérations de la boucle en temps – est représenté sur l'axe des ordonnées.

Les points de mesure sont accompagnés d'une barre d'erreur qui indiquent toutes un temps d'exécution très stable.

Les deux courbes des graphiques 8.9a et 8.9b représentent le temps d'exécution lorsque 10 threads OpenMP sont créés (5 par tâche MPI) – la puissance CPU est élevée – et lorsque 4 threads OpenMP sont créés (2 par tâche MPI) – la puissance CPU est moindre.

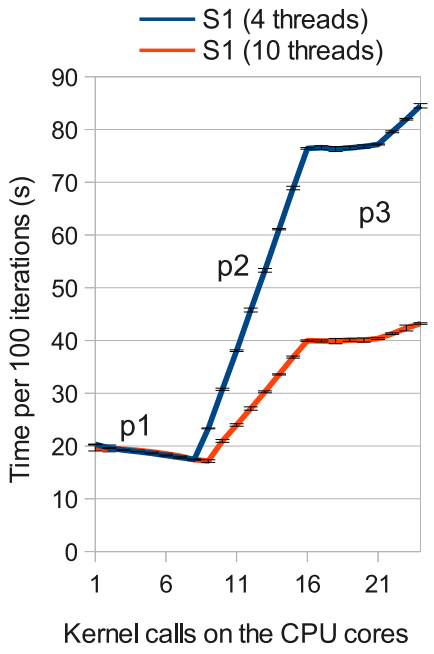
On peut isoler trois phases sur les courbes :

Description des trois phases :

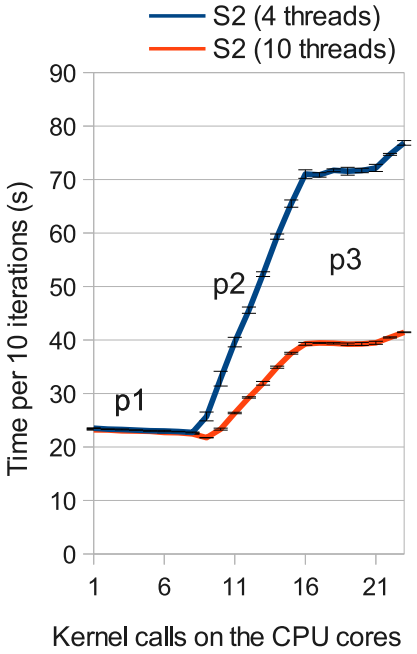
Phase 1 (p1) Le temps d'exécution des kernels GPU est plus important que celui de kernels CPU dans cette première phase. Ainsi, les calculs GPU dominent et prennent donc plus de temps que les calculs CPU. Par conséquent, lorsque l'on diminue le nombre de kernels GPU, on réduit le temps d'exécution. La puissance CPU a ici peu d'influence et c'est pour cette raison que les deux courbes sont globalement identiques, quelque soit le nombre de threads utilisés.

Phase 2 (p2) est une phase de transition. Les calculs CPU commencent à prendre plus de temps que les calculs GPU. Le temps d'exécution augmente fortement. L'impact des CPU est ici bien visible, et la pente de la courbe qui représente une exécution avec 4 threads est la plus importante.

Phase 3 (p3) Dans cette phase, le temps d'exécution des kernels CPU est plus important que celui des kernels GPU et les calculs CPU dominent. Par conséquent, lorsque l'on ajoute des kernels CPU, le temps de calcul augmente. Les courbes sont ici plutôt stables. Contrairement à la phase 1, la configuration à 10 threads diminue fortement le temps d'exécution par rapport à celle avec 4 threads.



(a) Performance de la stratégie A



(b) Performance de la stratégie B

Figure 8.9 – Performance des stratégies A et B lorsque le nombre de kernels exécuté sur les CPU augmente.

	$S1$	$S2$
Transferts (s)	5.23	2.85
Fusion des données (s)	0.65	6.52
Par rapport à la version GPU :	+45%	+75%

Table 8.1 – Coût des opérations de transfert et de fusion lorsque 10 threads CPU et 9 kernels sur les CPU sont utilisés).

Analyse Sans surprise, pour les phases 2 et 3, qui sont dominées par le calcul CPU, les performances sont meilleures lorsqu'un plus grand nombre de threads est utilisé. Le point d'équilibre entre le temps de calcul CPU et GPU a donc lieu à la fin de la phase 1 : c'est à ce moment que les temps de calcul CPU et GPU sont identiques. Ce point correspond au minimum de la courbe et indique la répartition des kernels qui offre les meilleures performances.

Bien que le temps d'exécution soit globalement identique avec 4 ou 10 threads durant la phase 1, nous observons qu'avec 10 threads, la phase 1 prend fin lorsque 9 kernels sont exécutés sur le CPU alors qu'avec 4 threads, elle s'arrête lorsque 8 kernels y sont exécutés. Ainsi, avec 10 threads, le point d'équilibre est légèrement plus bas qu'avec 4 threads, les performances de la parallélisation hybride sont donc meilleures lorsque la puissance CPU est la plus importante.

Ainsi, ajouter de la puissance CPU fait reculer le point d'équilibre et augmente par conséquent les performances globales de l'approche, que ce soit pour la stratégie A que pour la stratégie B.

8.3.3 Transferts de données et opérations de regroupement

Les valeurs représentées sur les graphiques 8.9a et 8.9b contiennent le temps de calcul mais également tous les temps de transferts et les opérations de mise à jour mémoire qui assurent la cohérence mémoire du calcul. Une analyse détaillée des exécutions des stratégies A et B, dont les résultats sont indiqués dans le tableau 8.1, nous permet de préciser quelle est la part des transferts et celle des calculs dans le temps total d'exécution.

Le tableau 8.1 indique le temps des transferts de données et le temps des opérations de fusion de données lorsque le temps de calcul CPU et GPU est équilibré, c'est-à-dire lorsque 9 kernels sont exécutés sur le CPU avec 10 threads OpenMP. Pour la stratégie A, la grosse quantité de données transférée amène un temps de transfert largement supérieur au temps de fusion des données. Pour la stratégie B, les optimisations réalisées sur les transferts réduisent fortement ces temps par rapport à A. Cependant, la complexité du code mis en œuvre pour regrouper les données partielles GPU et CPU augmente de manière très significative le temps de fusion.

Si on prend pour référence la version GPU de SPECFEM3D, les temps nécessaires aux opérations de cohérence de la stratégie A provoquent une augmentation de 45% du temps de calcul et ceux de la stratégie B une augmentation de 72% du temps de calcul. Ces temps sont très importants et empêchent la version SPECFEM3D/S_GPU 2 d'être plus performante que la version pure GPU de SPECFEM3D sur un nœud hybride.

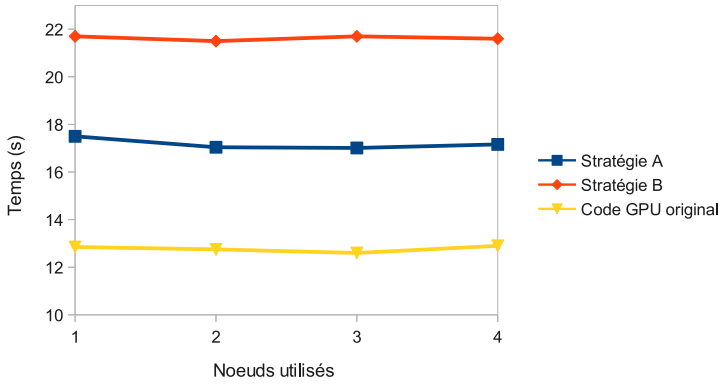


Figure 8.10 – Comparaison des stratégies A et B avec le code GPU original de SPECFEM3D sur 1, 2, 3 et 4 nœuds hybrides.

8.3.4 Évaluation sur une grappe hybride

Nous utilisons maintenant les quatre nœuds de notre grappe hybride afin de valider la scalabilité de la version SPECFEM3D/S_GPU 2 et également de confronter les performances de nos deux stratégies à la version originale purement GPU de SPECFEM3D. Sur chaque nœud, deux tâches MPI sont lancées. Nous allons évaluer les performances en faisant évoluer le nombre de tâches MPI pour une taille du problème fixe par tâche MPI. Ainsi, plus le nombre de tâches MPI sera important, plus la taille du système sera importante également. Sur la figure 8.10, sur laquelle nous avons indiqué les résultats, nous observons que la scalabilité de notre version S_GPU 2 est très bonne, identique à la version pure GPU de SPECFEM3D. Néanmoins, comme abordé dans la partie précédente, bien que la stratégie B réduise les transferts, les opérations de regroupement sont tellement coûteuses qu'elles entraînent un temps d'exécution supérieur à la stratégie A. De plus, la version purement GPU de SPECFEM3D montre des temps d'exécution meilleurs que nos stratégies A et B. Ainsi, dans notre cas, bien que l'utilisation des CPU ait une influence sur les performances, les opérations de maintien de la cohérence mémoire réduisent fortement les performances.

8.3.5 Comparaison des stratégies

La stratégie A est une solution relativement peu intrusive : les modifications du code source pour assurer la cohérence mémoire CPU et GPU sont très limitées et concernent uniquement quelques opérations basiques sur des tableaux de données. Cependant, elle transfère des données inutiles. Nous avons suggéré une autre solution qui demande des changements plus profonds dans l'application et une connaissance beaucoup plus poussée de son fonctionnement.

Ces deux stratégies représentent deux approches lorsque l'on aborde les problèmes de mises à jour mémoire avec S_GPU 2. Nous les avons classifiées en deux catégories.

— La première catégorie contient les stratégies ayant pour objectif de minimiser les modifications profondes des applications. Ainsi, les opérations prévues pour sélec-

tionner les données à transférer et les opérations destinées à fusionner les données des accélérateurs et des CPU sont simples. Dans des codes complexes, ces stratégies peuvent signifier transférer de grandes zones de mémoire et ainsi transférer des données non modifiées.
La stratégie A appartient à cette catégorie : un code simple est ajouté mais des transferts non optimums sont effectués.

— La seconde catégorie contient les stratégies qui proposent de réduire le plus possible les transferts de données ; les zones de données partielles modifiées par un calcul doivent être déterminées, transférées et fusionnées. La stratégie B appartient à cette catégorie et du code complexe doit être écrit pour sélectionner les données à transférer et à fusionner.

Choisir une stratégie appartenant à l'une ou l'autre de ces catégories dépend principalement de l'application avec laquelle S_GPU 2 est utilisée : la taille des données à transférer, la complexité du code à ajouter doivent être considérées. La qualité de l'équilibrage de charge entre CPU et GPU doit également être prise en compte. Ainsi, des outils adaptés à notre modèle de programmation sont nécessaires pour comprendre l'exécution d'un calcul hybride. Le système d'enregistrement des traces d'exécution implanté dans S_GPU 2 est une approche pour répondre à ce besoin.

8.4 Observation des exécutions de calculs hybrides

Comme S_GPU 1, S_GPU 2 supporte l'enregistrement de traces, qui, une fois interprétées par le logiciel de visualisation Vite, permet à un utilisateur de suivre le comportement des calculs hybrides d'une application S_GPU 2. Lors de la phase d'analyse de performances de SPECFEM3D, nous nous sommes concentrés sur le temps d'exécution global. Les traces nous ont servi à vérifier si les ressources étaient exploitées correctement.

Description des données observables Nous illustrons les capacités de traçage de S_GPU 2 avec SPECFEM3D. Les figures 8.11 et 8.12, montrent une interprétation des traces, avec Vite, d'une exécution de SPECFEM3D comportant 8 tâches MPI. Cela permettra de démontrer comment visualiser une exécution hybride S_GPU 2 comportant un grand nombre de tâches MPI. Pour faciliter l'observation, chaque tâche MPI est associée à une couleur. Chaque élément d'un calcul hybride est représenté avec la couleur de la tâche MPI ayant soumis ce calcul. Il est donc possible de déterminer par quelles tâches MPI les ressources CPU et GPU sont utilisées.

Durant la phase d'observation, les GPU virtuels gérés par S_GPU 2 sont représentés. Ces GPU virtuels sont donc visibles sur la figure 8.12 : comme les tâches MPI lancent des noyaux de calculs indépendants, plusieurs GPU virtuels sont simultanément exploités. Néanmoins, parce que nous utilisons uniquement un GPU physique ne supportant pas l'exécution concurrente de kernel (génération TESLA), l'exécution est sérialisée sur le GPU, sauf dans le cas des transferts mémoire où deux GPU virtuels peuvent faire un calcul et un transfert simultanément.

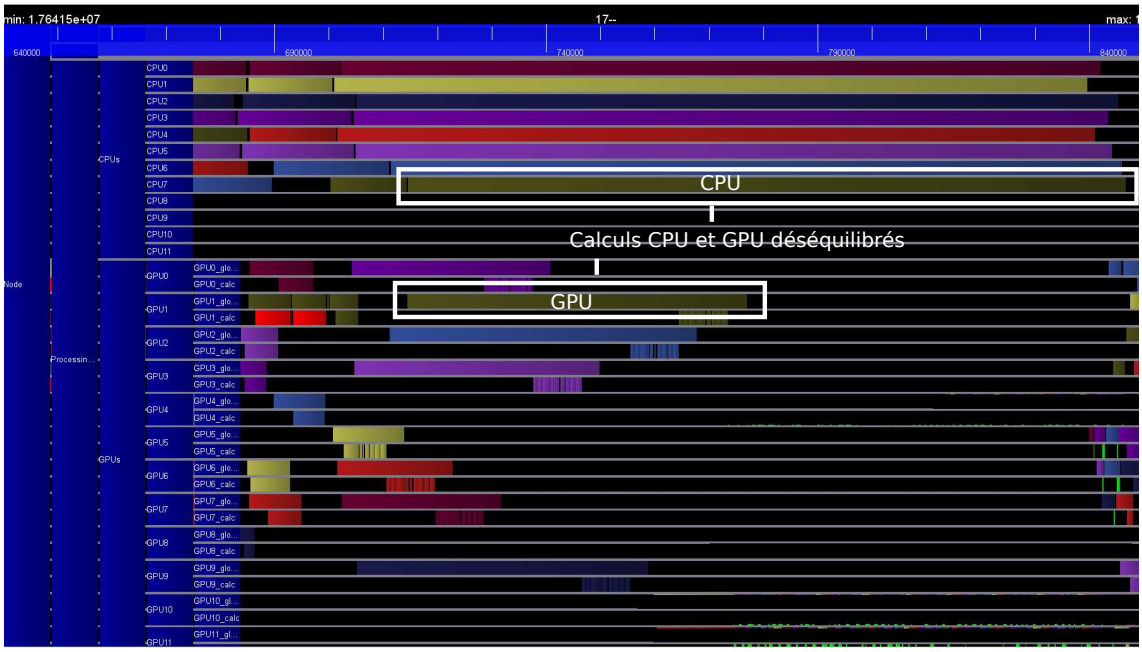


Figure 8.11 – Observation d'un déséquilibre entre CPU et GPU avec SPECFEM3D

Exécution déséquilibrée Avec la figure 8.11, il est possible d'évaluer l'utilisation des ressources CPU et GPU pour le calcul hybride de la partie 2 de la boucle en temps de SPECFEM3D. Nous avons 8 tâches MPI lancées, chacune d'entre elles utilise le GPU. Nous avons mis en évidence sur la figure 8.11, les parties CPU et GPU de l'étape 2 de la version hybride de la boucle en temps de SPECFEM3D. Sur cet exemple, les calculs GPU durent moins longtemps que ceux exécutés sur le CPU. Le calcul hybride est donc ici déséquilibré et la distribution des kernels doit être modifiée pour affecter plus de calculs à la partie GPU.

Exécution équilibrée La distribution des kernels a été modifiée, et nous constatons, sur la figure 8.12 que les temps d'exécution des parties CPU et GPU du calcul hybride sont proches : le calcul hybride est ici équilibré.

Les mécanismes de traçages intégrés à S_GPU 2 sont donc utiles pour vérifier l'équilibrage de charge entre CPU et GPU.

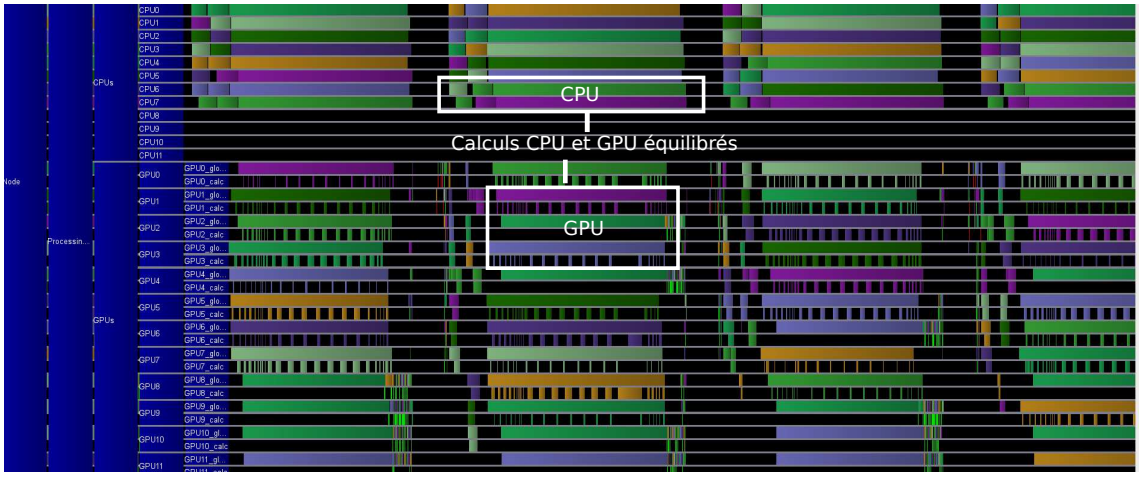


Figure 8.12 – Observation d'une exécution équilibrée

8.5 Conclusion

Nos expériences ont montré que le modèle de programmation hybride que nous avons proposé est adapté à des applications scientifiques existantes comme SPECFEM3D. Adapter un code pour S_GPU 2 afin qu'il exploite des ressources CPU et GPU pour un calcul est tout à fait possible, mais demande toutefois une bonne connaissance du fonctionnement de l'application.

Dans nos expérimentations, nous avons cherché à montrer quelle était l'influence de l'ajout de ressources CPU sur le temps d'exécution global de la version GPU de SPECFEM3D. Nous avons évalué deux stratégies pour maintenir la cohérence mémoire entre CPU et GPU. Plus la sélection des données à transférer est fine, plus le code de maintien de la cohérence mémoire est complexe. Au contraire, plus la sélection des données est lâche, plus le maintien de la cohérence mémoire est simple à mettre en place. Dans un cas comme dans l'autre, les opérations destinées à garantir la cohérence des mémoires CPU et GPU sont très lourdes et rendent la version hybride de SPECFEM3D moins performante que la version pure GPU.

Nous nous sommes concentrés sur cette évaluation à saisir quelles étaient les approches pour mettre en place un calcul hybride dans un code existant. Dans les expériences menées, la répartition des calculs entre CPU et GPU a été faite de manière statique en modifiant manuellement la répartition des appels de kernels de SPECFEM3D entre les CPU et les GPU. Nous n'avons pas évalué notre mécanisme d'équilibrage automatique de la charge implantée dans S_GPU 2. Cette évaluation devra être faite en travaux futurs. Il faut cependant noter que différents travaux [KTS10, LHK09], dont l'approche d'équilibrage est similaire à la nôtre, donnent de bons résultats.

Nous avons étudié S_GPU 2 avec SPECFEM3D. D'autres codes, nécessitant des opérations moins complexes lors de la mise en place de la cohérence mémoire CPU - GPU, pourraient tirer parti de manière plus efficace de l'approche calcul hybride de S_GPU 2. Les mises à jour mémoire sont les opérations qui diminuent les performances des calculs hybrides dans SPECFEM3D. Ces mises à jour sont nécessaires car les architectures

d'accélérateurs actuelles ont des zones mémoire disjointes. Des architectures nouvelles, comme les AMD Fusion [Adv11], qui possèdent un espace mémoire CPU et GPU unifié diminueraient – voire élimineraient – le besoin de codes complexes pour assurer la cohérence mémoire. Évaluer les performances sur ce type d'architecture nous permettrait donc d'avoir de nouvelles données concernant la pertinence de notre approche de calcul hybride.

Les nouveaux co-processeurs Intel de type Xeon Phi peuvent également apporter de nouvelles pistes. Ils sont en effet basés sur une architecture x86 classique et Intel a quasiment entièrement adapté sa suite logicielle pour cette architecture. Il est donc possible d'avoir un même code parallèle, par exemple en OpenMP, exécutable sur la machine hôte et sur les Xeon Phi. La création des threads CPU et accélérateur s'en trouverait ainsi considérablement simplifiée. De plus, des mécanismes avancés de partage de mémoire entre Xeon Phi et machine hôte devraient également simplifier la mise en place de stratégies de cohérence mémoire.

Chapitre Neuf

Conclusion et perspectives

Les machines multicœurs deviennent prépondérantes dans les supercalculateurs pour le calcul haute performance. L'efficacité énergétique de ces technologies multicœurs n'est plus à démontrer et sera une des clefs de la réussite du passage à l'exaflop.

Il existe différents types d'architectures multicœurs. Certaines sont constituées d'une dizaine de cœurs relativement puissants (comme les Xeon actuel). D'autres possèdent plusieurs centaines de cœurs très simples. C'est clairement cette voie qui a été choisie pour les accélérateurs : cette multiplication des cœurs offre une grande puissance et une excellente efficacité énergétique.

Les architectures à base d'accélérateurs prennent de plus en plus de place dans le calcul haute performance moderne. De plus en plus de machines sont hybrides ; elles associent CPU et accélérateurs. Tirer parti de toute la puissance disponible de ces machines hybrides est donc crucial pour exploiter complètement les supercalculateurs modernes. L'ajout d'accélérateurs de type GPU, qui ont leurs propres mémoires et une capacité d'exécution parallèle avec les CPU généralistes, pose des problème d'exploitation et également de programmation par rapport aux architectures classiques.

Grâce aux évolutions logicielles et matérielles de ces dernières années, les modèles de programmation pour accélérateurs deviennent de moins en moins contraints. Ils peinent cependant à être vraiment adaptés aux grandes applications parallèles. Différentes propositions ont eu lieu, comme par exemple des annotations dans le code source ou l'adaptation de modèles de tâches pour architectures hybrides.

Nous nous sommes intéressés dans cette thèse à des problématiques qui apparaissent lorsque des modèles de programmation très utilisés dans les applications scientifiques, comme MPI ou OpenMP, sont employés dans les supercalculateurs hybrides actuels. Ainsi, les problématiques de partage d'accélérateurs et d'exécution concurrente CPU - GPU ont été au cœur de cette thèse.

Nous avons montré, tout au long de ce document, comment nous avons pris soin de proposer des stratégies de programmation et d'exploitation qui puissent être intégrables dans les applications existantes – la notion d'intrusivité a été centrale dans cette thèse. Nous avons ainsi cherché à minimiser l'intrusivité. Nous avons également validé nos propositions sur deux grandes applications scientifiques.

Contributions

Les architecture hybrides sont très hétérogènes : le nombre d'accélérateurs et de cœurs CPU n'est pas lié. Ainsi, nous avons proposé un support exécutif destiné à exposer aux logiciels une architecture homogénéisée, c'est-à-dire qui possède un ratio nombre de

cœurs / nombre d'accélérateurs constant. Nous avons proposé pour cela une notion de virtualisation d'accélérateurs. Cette virtualisation permet de donner l'illusion aux applications qu'elles ont la capacité d'utiliser un nombre d'accélérateurs qui n'est pas lié au nombre d'accélérateurs physiques disponibles dans le matériel.

C'est la bibliothèque S_GPU 1 qui est chargée de mettre en place cette virtualisation en partageant les accélérateurs entre plusieurs processus MPI.

Lors de la validation de l'approche avec BigDFT, nous avons montré que la répartition homogène avec un partage de GPU offrait de bonnes performances avec une intrusivité limitée dans l'application. Des outils de visualisation adaptés, permettant de mettre en évidence les éventuels déséquilibres de charge, sont indispensables et ont été exploités dans nos travaux.

Nous avons également proposé des extensions aux modèles de programmation basés sur MPI et des threads afin de traiter le problème de l'exécution concurrente entre CPU et accélérateurs. Nous avons proposé un modèle basé sur deux types de threads, les threads CPU et accélérateur, permettant de mettre en place des calculs hybrides exploitant les CPU et les accélérateurs. C'est la bibliothèque S_GPU 2 qui est chargée de déployer, sur le matériel hybride, le parallélisme exprimé avec l'aide de ces threads.

La validation réalisée sur SPECFEM3D a montré que la gestion de la cohérence des différentes zones mémoire est cruciale en terme de performance. Nous avons évalué deux stratégies pour la mise en place de cette cohérence ; les deux sont extrêmement coûteuses. Cependant, un modèle de programmation basé sur des threads accélérateur et CPU est tout à fait intégrable dans une application scientifique existante.

Perspectives

Les perspectives ouvertes par nos travaux peuvent s'articuler autour de deux pistes. La première concerne la coopération entre support exécutif et compilateur, la seconde est liée au rapprochement des mémoires CPU et accélérateur.

- Coopération entre supports exécutifs et compilateurs La création des threads accélérateurs S_GPU 2 est basée sur une API assez bas niveau, proche d'une solution comme les Posix Threads. Cependant, les codes destinés à la simulation scientifiques préfèrent une approche plus haut niveau, comme les directives de compilation, ce qui est beaucoup plus simple à mettre en place.

Dans ce contexte, il pourrait être envisageable de créer directement à partir d'un même code source, des threads CPU et accélérateurs pour S_GPU 2. Des mécanismes de parallélisation automatique, comme OpenACC ou comme la nouvelle norme OpenMP 4, pourraient être envisagés pour générer automatiquement nos différents types de threads en y incorporant soit du code CPU soit du code accélérateur.

Ces stratégies enlèveraient l'étape de création manuelle des threads accélérateur. Il resterait cependant encore la problématique de la cohérence mémoire à traiter. En effet, les informations disponibles à la compilation sont assez statiques ; pour mettre en place notre approche de cohérence basée sur des fonctions de coupe et de regroupement, des informations dynamiques devront être fournies par le programmeur.

Ainsi, un support exécutif, qui serait une extension de S_GPU 2 pourrait déployer et exécuter quelques threads OpenMP sur le processeur central, et d'autres sur les accélérateurs, tout en assurant la cohérence mémoire de manière dynamique grâce aux informations du programmeur.

De nouvelles possibilités, comme la création de zones mémoire partagées entre CPU et accélérateurs, ou les prochaines possibilités des zones mémoire unifiées de CUDA pourraient permettre de simplifier la mise en place de cette cohérence en s'appuyant sur des mécanismes bas niveau plutôt que sur des mouvements de données explicites.

- Exploiter le rapprochement mémoire CPU - mémoire accélérateur Les machines hybrides évoluent rapidement ; depuis les premières cartes NVIDIA supportant uniquement la simple précision jusqu'aux derniers KEPLER ou Xeon Phi, de nombreuses évolutions technologiques ont eu lieu.

La mise en place des mouvements de données est une problématique centrale lorsque l'on souhaite exploiter les architectures hybrides. Ces mouvements complexifient énormément le développement de programmes hybrides et demandent des optimisations avancées (recouvrement, minimisation des transferts...) pour maximiser les performances. Ainsi, beaucoup de travaux sont réalisés pour rapprocher la mémoire centrale et la mémoire des accélérateurs. Un rapprochement logiciel des zones mémoire, plus ou moins abouti, est dès maintenant disponible, avec les technologies UVA de NVIDIA ou MYO pour les Xeon Phi d'Intel. Ce rapprochement est également effectué de manière matérielle : les processeurs *fusion* d'AMD ont une mémoire commune CPU et accélérateur. Il existe déjà des dispositifs qui réunissent des processeurs ARM et des accélérateurs partageant une même mémoire. Les projets de NVIDIA consistant à associer des processeurs ARM et des GPU, avec une mémoire commune ont également une proposition en ce sens.

Ces évolutions technologiques auront donc un impact important sur l'exploitation des architectures hybrides. Une zone mémoire unifiée entre CPU et accélérateur modifiera fortement la gestion des mouvements de données et serait donc un bond en avant considérable concernant la programmabilité de ces architectures. De plus, en supprimant la nécessité de gérer la cohérence de deux zones mémoire, ces évolutions pourraient très certainement augmenter les performances de notre approche de calcul hybride CPU – accélérateur.

Bibliographie

[AAD+10] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, 09 2010.

[AAD+11a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Egypt, 2011.

[AAD+11b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, 5 2011.

[ABI+09] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[Adv11] Advanced Micro Devices. *The AMD Fusion Whitepaper*, 2011.

[AMD09] AMD, http ://developer.amd.com. *AMD64 Architecture Programmer's Manuals*, 2009.

[Ash01] Peter J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001.

[ATI11] *AMD APP SDK Getting Started Guide (v2.4)*, 2011.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23 :187–198, February 2011.

[Aug09] Cédric Augonnet. StarPU : un support exécutif unifié pour les architectures multicœurs hétérogènes. In *19ème Rencontres Francophones du Parallélisme*, Toulouse / France, September 2009. Best Paper Award.

[BBB+08] Robert Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew, Andrew McCormick, Graham Smart,

Ronnie Smart, Allan Cantle, Richard Chamberlain, and Gildas Genest. Maxwell - a 64 fpga supercomputer. *Engineering Letters*, 16(3) :426–433, 2008.

[BDS96] Arndt Bode, Jack Dongarra, Thomas Ludwig 0002, and Vaidy S. Sunderam, editors. *Parallel Virtual Machine - EuroPVM'96, Third European PVM Conference, München, Germany, October 7-9, 1996, Proceedings*, volume 1156 of *Lecture Notes in Computer Science*. Springer, 1996.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. *ACM Trans. Graph.*, 23 :777–786, August 2004.

[BGGT01] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology Journal*, 2001.

[BHS+95] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. 1995.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[BvdSvD95] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs : A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3) :43 – 56, 1995.

[BZS10] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Automatic vector instruction selection for dynamic compilation. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 573–574, New York, NY, USA, 2010. ACM.

[CAO+06] Alberto Castro, Heiko Appel, Micael Oliveira, Carlo A. Rozzi, Xavier Andrade, Florian Lorenzen, M. A. L. Marques, E. K. U. Gross, and Angel Rubio. octopus : a tool for the application of time-dependent density functional theory. *phys. stat. sol. (b)*, 243(11) :2465–2488, 2006.

[CE00] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[CH03] L. V. Kale Chao Huang, Orion Lawlor. Adaptive mpi. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, LNCS 2958 :306–322, 2003.

[cha11] Chapel language specification, April 2011.

[CKL+08] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa Tikir, David Michéa, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62 thousand processor cores. In *SC'08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Austin, Texas, USA, November 2008. Article #60, Gordon Bell Prize finalist article.

[cly10] Clyther. clyther : Python language extension for opencl., 2010.

[cod]

[CR75] Inc Cray Research. An introduction to the cray-1 computer. Technical report, Cray Research, Inc, 1975.

[CR82] Inc Cray Research. The cray x-mp series of computer systems. Technical report, Cray Research, Inc, 1982.

[CR85] Inc Cray Research. The cray-2 computer system. Technical report, Cray Research, Inc, 1985.

[DBB07] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : a hybrid multicore parallel programming environment. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[dKSB00] J. Chassin de Kergommeaux, B. Stein, and P. E. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10) :1253 – 1274, 2000.

[Dro08] Paul J. Drongowski. *Basic Performance Measurements for AMD AthlonTM 64, AMD OpteronTM and AMD PhenomTM Processors*. AMD, September 2008.

[DY08] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony : an execution model and runtime for heterogeneous many core systems. In *HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, New York, NY, USA, 2008. ACM.

[DY10] Gregory Diamos and Sudakhar Yalamanchili. Speculative execution on Multi-GPU systems. In *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[Fat09] Massimiliano Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, New York, NY, USA, 2009. ACM.

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21 :948+, 1972.

[Fre] Free Software Foundation, http ://gcc.gnu.org/projects/tree-ssa/vectorization.html. *Auto-Vectorization in GCC*.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GNG+08] Luigi Genovese, Alexey Neelov, Stefan Goedecker, Thierry Deutsch, Seyed Ghasemi, Alexander Willand, Damien Caliste, Oded Zilberberg, Mark Rayson, Anders Bergman, and Reinhold Schneider. Daubechies wavelets as a basis set for density functional pseudopotential calculations. *Journal of Chemical Physics*, 129, 2008.

[GOD+09] Luigi Genovese, Matthieu Ospici, Thierry Deutsch, Jean-François Méhaut, A. Neelov, and S. Goedecker. Density Functional Theory Calculation on many-cores Hybrid CPU-GPU architectures. *Journal of Chemical Physics*, 131(3) :034103, jul 2009.

[GOV+11] Luigi Genovese, Matthieu Ospici, Brice Videau, Thierry Deutsch, and Jean-François Méhaut. *GPU Computing Gems*, chapter Wavelet-based Density Functional Theory Calculation on Massive Parallel Hybrid Architectures. Moran Kaufmann, jan 2011.

[gtg] *Generic Trace Generator*. https ://gforge.inria.fr/projects/gtg/.

[GVO+11] Luigi Genovese, Brice Videau, Matthieu Ospici, Thierry Deutsch, Stefan Godecker, and Jean-François Méhaut. Daubechies Wavelets for High Performance Electronice Structure Calculations : the BigDFT Project. *Comptes Rendus de l'Académie des Sciences*, 339(2) :149–164, feb 2011. Special Issue on Intensive Computing. Editor : O. Pironneau.

[Hen07] Justin Hensley. Amd ctm overview. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[Hov08] Rune Johan Hovland. Latency and bandwidth impact on gpu-systems, December 2008.

[IB00] The infiniband architecture specification, 2000.

[IBM08] IBM. *IBM SDK for Multicore Acceleration v.3.1*, 2008.

[INR10] INRIA. *ViTE User Manual*, september 2010.

[Int11a] Intel. *Intel Math Kernel Library Reference Manual 10.3*, 2011.

[INT11b] INTEL. *Intel Trace Analyser and Collector User's Guide.*, 2011.

[INT11c] INTEL. *Intel Trace Collector User's Guide.*, 2011.

[INT11d] INTEL. *Intel VTune Amplifier XE 2011 Help for Linux* OS*, 2011.

[Int11e] Intel, http ://www.intel.com/products/processor/manuals/. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B : System Programming Guide, Part 2*, 2011.

[Int12] Intel. Intel® 64 and ia-32 architectures software developer's manual. Technical report, Intel, 2012.

[JB94] B. Morillon et J.C. Nimal J.P. Both, H. Derriennic. Tripoli-4. *Proceedings of the 8th International Conference on Radiation Shielding (ICRS)*, 1 :373, 1994.

[Kal90] L.V. Kale. The chare kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, 1990.

[KEGM10] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20) :7692–7714, 2010.

[KPL+11] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL : A Scripting-Based Approach to GPU Run-Time Code Generation. *ArXiv e-prints*, March 2011.

[KTS10] C. Kartsaklis, I.T. Todorov, and W. Smith. DL POLY 3 : Hybrid CUDA/OpenMP porting of the non-bonded force-field for two-body systems. In *Symposium on Chemical Computations on GPGPUs, 240th ACS National Meeting and Exposition*, 2010.

[Lei85a] Charles E. Leiserson. Fat-trees : universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34 :892–901, October 1985.

[Lei85b] Charles E. Leiserson. Fat-trees : universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10) :892–901, October 1985.

[LHK+04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu : general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, September 1979.

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

[MGH+96] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling watchdog : combining polling and interrupts for efficient message handling. *SIGARCH Comput. Archit. News*, 24 :179–188, May 1996.

[MKJ+07] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[MN06] Allen D. Malony and Wolfgang E. Nagel. The open trace format (otf) and open tracing for hpc. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[MPI] MPICH : Implementation of MPI, http ://www-unix.mcs.anl.gov/mpi/mpich/.

[mpi09] Mpi : A message-passing interface standard, September 2009.

[NH06] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17 :1–31, August 1998.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. *SIGPLAN Not.*, 41 :132–143, June 2006.

[NVI] NVIDIA. Fermi compute architecture whitepaper.

[NVI11a] NVIDIA. *NVIDIA CUDA Programming Guide 4.0*, 2011.

[NVI11b] NVIDIA. Thrust quick start guide. Technical report, NVIDIA, 2011.

[OGD09] Matthieu Ospici, Luigi Genovese, and Thierry Deutsch. Exploitation et partage de GPU dans des grappes de calcul hybrides. In *Rencontres francophones du Parallélisme (RenPar'19)*, 2009.

[OKMD11] Matthieu Ospici, Dimitri Komatitsch, Jean-François Méhaut, and Thierry Deutsch. SGPU 2 : a runtime system for using of large applications on clusters of hybrid nodes. In *Second Workshop on Hybrid Multi-core Computing, held in conjunction with HiPC 2011*, Bangalore, India, dec 2011.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[omp08] OpenMP application program interface v.3.0, 2008.

[OPE11a] The opencl specification, version 1.1, 2011.

[Ope11b] OpenACC. The OpenACC application programming interface. Technical report, OpenACC, 2011.

[pap] Papi user's guide.

[PBW+05] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *J. Comput. Chem.*, 26(16) :1781–1802, December 2005.

[PGI11] Pgi cuda-x86, 2011.

[PM12] M. Pharr and W.R. Mark. ispc : A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1 –13, may 2012.

[Por10] The Portland Group. *PGI Fortran & C Accelerator Programming Model ver. 1.3*, 2010.

[pos95] Ieee std. 1003.1c-1995 thread extensions, 1995.

[PSG06] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for gpus. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[PTA+92] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. Iterative minimization techniques for *ab initio* total-energy calculations : molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64 :1045–1097, Oct 1992.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition : The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[Ram06] R.M. Ramanathan. Intel® multi-core processors making the move to quad-core and beyond. Technical report, Intel, 2006.

[Sei98] Rich Seifert. *Gigabit Ethernet : Technology and Applications for High-Speed LANs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[SFT+09] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne, and Daniel Etiemble. Algorithmic skeletons within an embedded domain specific language for the cell processor. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 67–76, Washington, DC, USA, 2009. IEEE Computer Society.

[SM06] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20 :287–311, May 2006.

[SOW+95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[SSH08] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. Languages and compilers for parallel computing. chapter MCUDA : An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[TKL08] Jeroen Tromp, Dimitri Komatitsch, and Qinya Liu. Spectral-element and adjoint methods in seismology. *Communications in Computational Physics*, 3(1) :1–32, 2008.

[TRF+11] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace : a generic framework for performance analysis. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, États-Unis, May 2011. Poster Session.

[upc05] Upc language specifications, 2005 2005.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, August 1990.

[vam] *Vampir user's guide*. http ://www.vampir.eu/.

[ver96] Verilog-a language reference manual, August 1996.

[VHHZ08] Duc Vianney, Gad Haber, Andre Heilper, and Marcel Zalmanovici. Performance analysis and visualization tools for cell/b.e. multicore environment. In *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, IFMT '08, pages 7 :1–7 :12, New York, NY, USA, 2008. ACM.

[Wes08] Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http ://-charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[WT11] Xingfu Wu and Valerie Taylor. Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers. *SIGMETRICS Perform. Eval. Rev.*, 38 :56–62, March 2011.