



ENSEIRB
Informatique
PRCD



CERFACS
CSG

Portage d'elsA sur GPU

Auteur :
Julien BRUNEAU

Maître de stage : Mme. Isabelle D'Ast
Tuteur de stage : M. Brice Goglin

Résumé

Ce rapport présente mon travail effectué au CERFACS, sur l'utilisation des cartes graphiques pour l'accélération du code elsA (calculs de mécanique des fluides). Les cartes graphiques présentent des capacités de calculs importantes dues à leur haut niveau de parallélisme. Le but du stage est d'étudier l'accélération que l'on peut obtenir sur ce code en utilisant les cartes graphiques. Pour cela, un cas test monobloc fut accéléré en portant 6 routines de calculs sur GPU. Ce portage accéléra l'exécution du cas test par 2. Mais les résultats ont permis de calculer une accélération de 7 sur le cas test, que l'on peut espérer obtenir en portant l'intégralité des routines du cas test. De plus, des tests ont été effectués sur le même cas test en multi-blocs puis avec l'utilisation de plusieurs GPU. Cela a permis de vérifier que les performances restaient valables en parallélisant l'usage des GPU.

Remerciements

Je tiens à remercier Mme Isabelle D'Ast, M. Nicolas Monnier ainsi que toute l'équipe CSG pour m'avoir accueilli pour se stage. Je tiens à les remercier aussi pour leur présence et leur aide ainsi que Louis Leredde, stagiaire et ancien camarade de classe qui partageait le même bureau que moi.

Je remercie aussi M. Jean-François Boussuge, M. Michel Gazaix et M. Marc Montagnac pour leur aide sur le code d'elsA. Je remercie Total pour leur invitation à la formation HMPP, M. F. Courteille de NVidia pour ses conseils et la hotline PGI pour leurs informations.

Enfin je remercie toutes les personnes du CERFACS qui se sont montrées très accueillantes et M. Brice Goglin, mon tuteur de stage, qui a fait preuve de beaucoup d'implication dans le suivi du stage.

Table des matières

1	Introduction	4
2	Présentation du CERFACS	6
2.1	Les différentes équipes	7
2.1.1	Algorithmique parallèle (Algo)	7
2.1.2	Dynamique des fluides (CFD)	7
2.1.3	Climat (GlobC)	8
2.1.4	Electromagnétisme	8
2.1.5	Aviation et environnement (AE)	8
2.1.6	Transfert de technologie	9
2.1.7	Computer Support Group (CSG)	9
3	Architecture GPU	10
3.1	Architecture de Base	10
3.1.1	Streaming Multiprocessors	10
3.1.2	Modèle d'exécution	12
3.2	La mémoire	14
3.2.1	La mémoire globale	14
3.2.2	Autre mémoire distante	14
3.2.3	La mémoire sur les multiprocesseurs	16
3.2.4	Classement des mémoire	16
3.3	FERMI	18
4	Les langages de développement	19
4.1	CUDA C	19
4.1.1	CUDA RUNTIME LIBRARY	20
4.1.2	CUDA Driver API	21
4.2	CUDA Fortran	23
4.3	Autres	25
5	Le projet elsA	27
5.1	Présentation	27
5.2	Structure du code	28
5.3	Compilation	28

6	Le portage	30
6.1	Portage des routines	30
6.1.1	Première stratégie de portage	30
6.1.2	Deuxième stratégie de portage	32
6.1.3	Optimisation	34
6.1.4	Portage en CUDA FORTRAN	36
6.2	Multi-Blocs et MPI	37
6.2.1	Multi-Blocs	38
6.2.2	Multi-GPU	38
6.3	Gestion de la mémoire	39
6.3.1	Les classes	39
6.3.2	Les transferts effectués	40
7	Résultats	44
7.1	Matériel pour les mesures	44
7.2	Performances	45
7.2.1	Par routines	45
7.2.2	Sur la totalité d'elsA	46
7.2.3	Potentiel	47
7.3	Multi-blocs	49
7.3.1	Mono GPU	49
7.3.2	Multi GPU	51
7.4	Validation des résultats	52
8	Conclusion	55
	Bibliographie	57
A	Profilings	58
B	Tableau des temps	62

Chapitre 1

Introduction

Ces dernières années, les constructeurs de processeurs se sont orientés vers les architectures multi-coeurs. En effet, à cause de problème de dissipation thermique et de consommation électrique, la fréquence des processeurs a cessé d'augmenter. Pour palier ce problème et tirer avantage de la place libre sur les processeurs, les constructeurs ont commencé à rajouter des coeurs sur leurs processeurs. Les cartes graphiques ont adopté ces architectures bien avant, car elles doivent traiter un grand nombre de données pour le rendu graphique de manière parallèle. Aujourd'hui, elles possèdent des architectures massivement parallèles ayant des pics de performance bien supérieurs aux CPU, comme on peut le voir sur la courbe 1.1. Jusqu'à il y a très peu de temps, développer des applications de calculs sur GPU était très difficile. Il fallait une bonne connaissance de la programmation graphique et utiliser un langage très bas niveau comparable à l'assembleur (le ptx chez NVidia). L'une des premières plateformes de programmation généralisée sur GPU (General Purpose Graphical Processor Unit), le modèle Brook, fut diffusée en 2004. Depuis la communauté du GPGPU s'est beaucoup développée, surtout dans le domaine du calcul scientifique. Il y a maintenant de nombreux langages permettant de bénéficier des performances des GPU ainsi que de nombreux exemples de codes ou de calculs portés sur GPU obtenant de bons facteurs d'accélération.

La programmation GPU est très différente de la programmation CPU ou multi-CPU. Le niveau de parallélisme y est bien supérieur, on passe de quelques centaines de threads et processus parallèles sur CPU à plusieurs milliers de threads sur GPU. Le passage d'une application CPU à une application GPU n'est pas forcément trivial. En plus de devoir traduire le code, il faut le paralléliser voire même changer l'algorithme pour que ce dernier soit mieux adapté au GPU. Sans oublier qu'il faudra ensuite mettre à jour les serveurs de calculs pour y ajouter des GPU. Enfin, on ne peut pas certifier qu'un code obtiendra les facteurs d'accélération attendus en passant sur GPU. L'architecture est très spécifique aux codes ayant beaucoup de calculs à effectuer en parallèle et les pertes de performances peuvent être importantes. C'est pourquoi il est

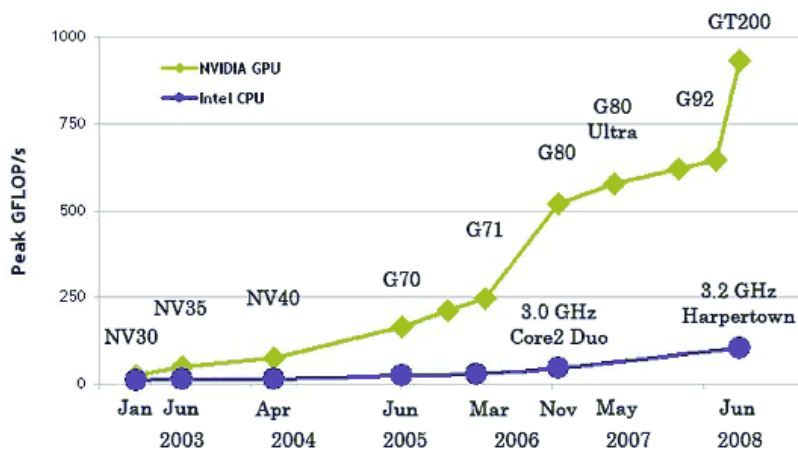


FIG. 1.1 – Évolutions des pics de calculs floatant des CPU Intel et des GPU NVidia (courbe provenant de la documentation NVIDIA)

préférable de prendre des précautions et de faire quelques tests avant de décider de porter entièrement ou une partie d'une application sur GPU.

Le stage est une seconde approche de portage de l'application elsA présentée dans le chapitre 5. En effet, un stage précédent ([3]) effectué à l'ONERA concernant elsA et les GPU a abouti au portage de 2 routines. Le stage actuel doit permettre de répondre à la question suivante : "Quelles sont les performances que l'on peut espérer obtenir avec elsA sur GPU?" De plus il fournira des informations sur la quantité de travail et le temps nécessaire au portage d'elsA sur GPU. Le stage doit permettre au CERFACS de décider s'il est rentable de continuer sur cette voie ou s'il est préférable de l'abandonner.

Chapitre 2

Présentation du CERFACS

Le CERFACS (Centre Européen de Recherche et Formation Avancées en Calcul Scientifique) est un centre de recherche privé à but non lucratif fondé en 1987. Il se situe à Toulouse, sur le site de Météo France. Les domaines de recherches sont nombreux allant de la météorologie à l'aéronautique en passant par l'électromagnétique. Pour cela, les chercheurs du CERFACS utilisent de puissants serveurs de calculs utilisant les dernières technologies disponibles. Le CERFACS est dirigé par un conseil de gérance constitué de ses 7 actionnaires :

- CNES (Centre national d'études spatiales),
- EADS (France European Aeronautic and Defence Space Company),
- EDF (Électricité de France),
- Météo-France,
- ONERA (Office national d'études et de recherches aérospatiales),
- SAFRAN,
- TOTAL.

Le CERFACS est composé d'une centaine d'employés dont plus de 90% sont des ingénieurs et des chercheurs de différents pays. Ils travaillent sur des projets répartis en 9 domaines de recherche principaux : l'algorithmique parallèle, le couplage de code, l'aérodynamique, les turbines de gaz, la combustion, le climat, l'impact environnemental, l'assimilation de données et l'électromagnétique. De plus le CERFACS collabore avec d'autres entités. Avec le CNRS (Centre National de la Recherche Scientifique, www.cnrs.fr), ils ont formé une Unité de Recherche Associée. Ils ont un laboratoire commun avec l'INRIA (Institut National de Recherche en Informatique et Automatique, www.inria.fr). De plus ils participent au TVE (Terre Vivante et Espace, www.omp.obs-mip.fr/tve), aux activités du AESE (Pôle de Compétitivité "Aéronautique, Espace et Systèmes Embarqués", www.aerospace-valley.com) et sont membres du RTRA/STAE ((Réseau Thématique de Recherche Avancée "Sciences et Technologies pour l'Aéronautique et l'Espace", www.fondation-stae.net/fr/), voir aussi www.cerfacs.fr/RTRA-STAE.mpg).

2.1 Les différentes équipes

2.1.1 Algorithmique parallèle (Algo)

Comme son nom l'indique, l'équipe Algo traite d'algorithmique parallèle. Son but est de développer des méthodes numériques génériques de résolution de systèmes d'équations, ainsi que des techniques d'optimisations liées à l'utilisation de machines parallèles. Toutes les autres équipes du CERFACS ayant recours à des calculateurs et des algorithmes parallèles, l'expertise de l'équipe Algo est très importante. Les recherches menées par cette équipe sont très variées :

- optimisation de coeurs de calculs et bibliothèques pour calculateurs hautes performances,
- calculs sur matrices creuses (pré-conditionneurs, solveurs directs et itératifs, ...)
- décomposition de domaines,
- systèmes non linéaires et optimisations,
- fiabilité et qualité des calculs numériques.

2.1.2 Dynamique des fluides (CFD)

L'équipe CFD (Computational Fluid Dynamics) couvre de nombreux domaines de la mécanique des fluides. Cette équipe a pour but de résoudre des problèmes mêlant dynamique des fluides et calculs hautes performances. Malgré les avancées technologiques dans ces deux domaines, certains problèmes restent toutefois impossibles à résoudre avec les moyens actuels. Les recherches effectuées par cette équipe visent donc un compromis entre exactitude physique et faisabilité numérique.

Les problèmes traités par cette équipe sont nombreux, et vont de l'aérodynamique classique à l'étude des turbulences, en passant par la combustion. En réalité, l'équipe CFD est subdivisée en deux entités principales (combustion et aérodynamique), chacune traitant d'un point particulier de la dynamique des fluides.

Combustion

La mission de CFD dans ce domaines est l'étude et la simulation de la combustion de carburant dans les moteurs aéronautiques et automobiles. Cela implique des simulations de flammes, de transfert de chaleur, des réactions chimiques, ainsi que des effets acoustiques. Ces études permettent d'une part la conception de moteurs plus performants, et d'autre part la prévision des émissions de polluants ainsi que des instabilités provoquées par la géométrie des moteurs.

Ces études nécessitent des simulations mêlant physique et chimie, et requièrent des outils spécifiques et performants. L'équipe CFD développe l'application AVBP (copropriété CERFACS-IFP) qui permet de simuler des phénomènes

complexes en utilisant diverses méthodes numériques issues de chacun des domaines scientifiques mis en jeu.

Aérodynamique

Le but de l'équipe CFD dans le domaine de l'aérodynamique est de développer des outils numériques qui puissent être utilisés aussi bien dans un milieu de recherche que dans l'industrie. La plupart des efforts sont portés vers le développement de l'application nommée elsA (Ensemble Logiciel pour la Simulation en Aérodynamique). On retrouvera une description de l'application au chapitre 5.

2.1.3 Climat (GlobC)

GlobC (GLOBAL Change) est une équipe du CERFACS faisant partie du CNRS au titre d'unité de recherche associée. L'équipe mène des activités de recherches théoriques et appliquées dans le domaine de la modélisation du climat. Son objectif principal est de faire évoluer la compréhension et les techniques de prévision de phénomènes liés à la dynamique du climat (océan, atmosphère) pour des échelles globales ou régionales. Pour réaliser cela, GlobC étudie les thèmes suivants :

- les variations climatiques (d'origines naturelles ou résultantes d'activités humaines),
- validation de systèmes de prévisions saisonnières reposant sur des assimilations de données d'observations océaniques,
- développement du logiciel PALM pour les problèmes de couplage d'applications d'assimilation de données,
- développement du logiciel OASIS pour le couplage de modèles océaniques et atmosphériques.

2.1.4 Electromagnétisme

Cette équipe a pour but de développer de nouvelles méthodes pour résoudre des problèmes utilisant les équations de Maxwell. Elle conçoit de nouveaux algorithmes utilisables pour des applications industrielles.

Ces recherches visent à améliorer les performances des radars, la conception des antennes ainsi qu'à calculer divers champs magnétiques. Ce domaine est très utile, que ce soit pour l'aérodynamique ou pour l'industrie en général.

2.1.5 Aviation et environnement (AE)

Que ce soit par émissions directes de produits chimiques, ou par la modification du bilan radiatif de l'atmosphère, les avions ont un impact sur l'environnement. Afin d'étudier cela et d'en réduire les nuisances, l'équipe AE quantifie les modifications locales et globales de l'atmosphère dues à ces phénomènes. A

cet effet, des modélisations numériques sont réalisées. Parmi les outils utilisés, on peut compter AVBP et NTMIX, deux codes développés par le CERFACS, ainsi que divers autres codes du Centre National de Recherche en Météorologie (Més0-NHC, Arpège, MOBIDIC et MOCAGE).

2.1.6 Transfert de technologie

L'équipe transfert de technologie a deux activités principales. La première concerne la simulation des flux d'air et thermiques. Elle propose des simulations numérique du flux d'air à l'intérieur de bâtiment pour prédire la qualité de l'air et le confort thermique. Cela permet d'analyser les systèmes de chauffages, de ventilations et de climatisations et d'optimiser la qualité de l'air et le confort. La deuxième activité, concernant les solutions de collaboration, propose des outils basés sur les nouvelles technologies, permettant de travailler à distance. La première version disponible permet à plusieurs collaborateurs d'effectuer l'analyse des résultats de simulations à distance comme s'ils étaient dans la même pièce. Cette version intègre également, par l'intermédiaire d'une interface graphique intuitive, la rédaction de documents en collaboration et le partage d'écran ainsi que le partage des fonctionnalités de manipulation des données 3D.

2.1.7 Computer Support Group (CSG)

Pour mener à bien leurs recherches, toutes les équipes précédentes ont besoin de calculateurs et d'interconnexions performants. Pour cela le CERFACS dispose de l'équipe CSG, mon équipe d'accueil, dont les buts sont les suivants :

- définitions de l'architecture globale de tous les moyens informatiques du CERFACS, incluant les infrastructures liées aux technologies de l'information ainsi que l'anticipation des besoins dans ces domaines
- intégration de solutions et services permettant de répondre aux attentes des utilisateurs,
- support aux utilisateurs en offrant et en maintenant les outils liés au calcul scientifique et en leur apportant une expertise en termes de développement et optimisation de code,
- partage de connaissances.

Chapitre 3

Architecture GPU

On ne décrira dans cette partie que les architectures des cartes graphiques NVidia. Les outils de développement et la communauté sont bien plus importants du coté NVidia que du coté ATI. D'ailleurs la quasi totalité des serveurs de calculs intégrant des GPU sont constitués de carte NVidia. C'est pourquoi le portage a été effectué sur des cartes NVidia. On commencera par décrire les architectures de versions inférieures à 2.0. Puis on verra les optimisations apportées par la nouvelle architecture FERMI, correspondant à la version 2.0, qui est sortie pendant mon stage. Pour plus d'information voir le manuel de NVidia [1].

3.1 Architecture de Base

La plus grosse différence entre les CPU et les cartes graphiques est le nombre de coeurs de calcul. Un CPU est composé de 1 à 12 coeurs de calculs. Une carte graphique est composée d'une trentaine de Multiprocesseurs contenant chacun jusqu'à 8 coeurs de calculs (pour les version < 2.0). Ce qui lui permet d'obtenir des pics de performances beaucoup plus importants que ceux des CPU. En étudiant le schéma 3.1 de l'architecture GPU on remarque deux parties. En bas du schéma on a la mémoire de la carte et en haut les multiprocesseurs appelés Streaming Multiprocessors (SM).

3.1.1 Streaming Multiprocessors

- Les multiprocesseurs sont composés de :
- 8 coeurs de calculs dont 1 double précision,
 - 1 unité de contrôle,
 - registres qui seront répartis entre les threads,
 - mémoire partagée,
 - et cache pour certaines mémoires optimisées.

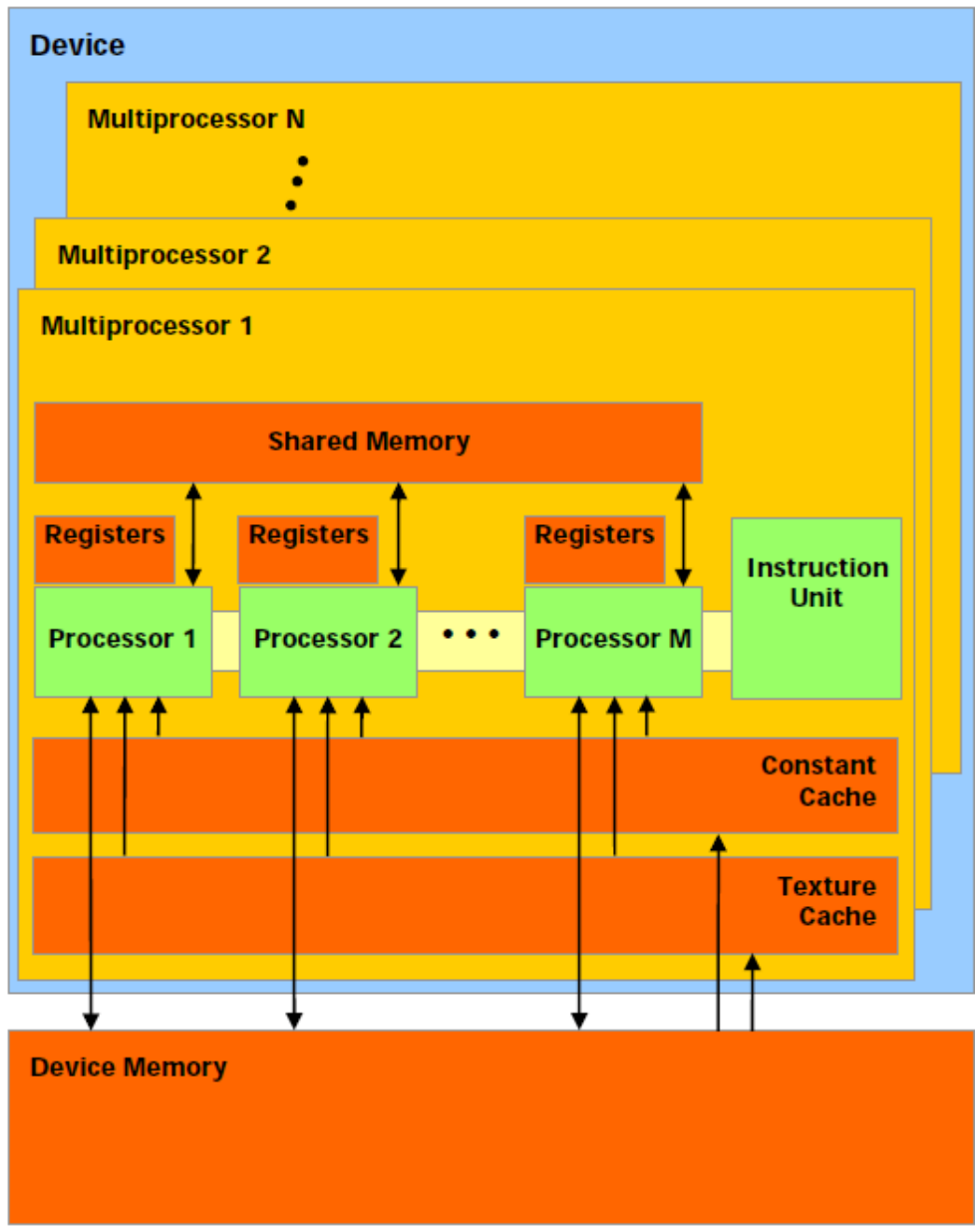


FIG. 3.1 – Architecture GPU (figure provenant de la documentation NVIDIA)

L'unité de contrôle lit les instructions et les envoie aux 8 coeurs de calculs qui exécutent les instructions de base en 4 cycles d'horloge. Cela signifie que les 8 coeurs de calculs exécutent les mêmes instructions en même temps ou qu'un groupe de threads est exécuté en parfaite synchronisation. Cela permet d'ailleurs, dans certains cas, d'éviter l'utilisation de fonctions de synchronisation entre les threads et ainsi de gagner en performances. De plus, lorsque qu'une condition, par exemple un if/else, est exécutée le groupe de threads exécute cette condition en même temps. Si 90% des threads du groupe doivent exécuter les instructions du if et les 10% restant doivent exécuter les instructions du else, l'unité de contrôle devra lire toutes les instructions. Dans la partie if, 10% des threads seront inactifs et dans la partie else 90% des threads seront inactifs. Alors que, si tous les threads doivent exécuter la même partie de la condition, l'unité de contrôle ne lira que les instructions correspondantes. Le premier cas est un phénomène appelé divergence de branch qui peut être mesuré par le profiler NVidia et qui doit être évité au maximum pour ne pas faire chuter les performances.

On vient de voir qu'un groupe de threads exécutait les mêmes instructions en même temps. Ayant 8 coeurs de calculs sur les multiprocesseurs, on a vu que 8 threads exécutaient les mêmes instructions en même temps. En réalité les instructions sont exécutées par groupe de 32 threads. Les coeurs de calculs ont un très haut niveau de pipeline. C'est à dire que l'instruction suivante commence à être traitée avant que le traitement de la précédente soit terminé. Ainsi pour une latence de 4 temps d'horloge pour l'exécution d'une instruction on a un débit de 1 temps d'horloge. NVidia a rassemblé les $4 * 8$ threads qui sont exécutés en un cycle d'instruction (4 temps d'horloge) pour en faire un warp. Ainsi les 32 premiers threads forment un warp, les 32 suivants en forment un autre, etc.

3.1.2 Modèle d'exécution

Lors de l'exécution d'un noyau de calcul sur GPU, le développeur définit une géométrie. C'est à dire que les threads sont répartis dans des blocs. Ainsi les multiprocesseurs exécutent un ou plusieurs blocs de thread (voir schéma 3.2). La taille des blocs est soumise à optimisation. Premier point, il est préférable d'avoir des blocs de taille multiple de 32 ou de 16, 32 pour éviter d'avoir un warp incomplet dans le bloc et 16 pour les optimisations d'accès mémoire abordées dans la section suivante. Ensuite un bloc ne peut pas être découpé sur plusieurs multiprocesseurs, il doit pouvoir tenir entièrement sur l'un d'eux avant d'être exécuté. Le système va bien sur essayer de mettre un maximum de blocs sur les multiprocesseurs comme le montre le schéma 3.2. Mais il y a certains facteurs limitants :

- il y a une limite de 6 blocs par multiprocesseurs,
- le nombre de registres utilisés par les blocs d'un multiprocesseur doit être inférieur ou égal au nombre de registres présents sur le multiprocesseurs,
- la quantité de mémoire partagée utilisée par les blocs d'un multipro-

cesseurs doit être inférieure ou égale à la quantité de mémoire partagée que possède le multiprocesseurs.

Cela signifie que tous les blocs ne seront pas toujours distribués sur les multiprocesseurs car ces derniers seront pleins. Ils attendront la fin de l'exécution d'un autre bloc pour prendre sa place. On va alors chercher à optimiser le nombre de threads pouvant être exécutés en même temps par un multiprocesseur en jouant sur ces différentes variables : nombre de registres utilisés par un thread, quantité de mémoire partagée utilisée et le nombre de threads par bloc. NVidia fournit une feuille excel permettant de calculer le taux d'occupancy en fonction des ces différentes variables. Le taux d'occupancy est le rapport du nombre de threads exécutés par multiprocesseur sur le nombre maximum de threads que peut traiter un multiprocesseur.

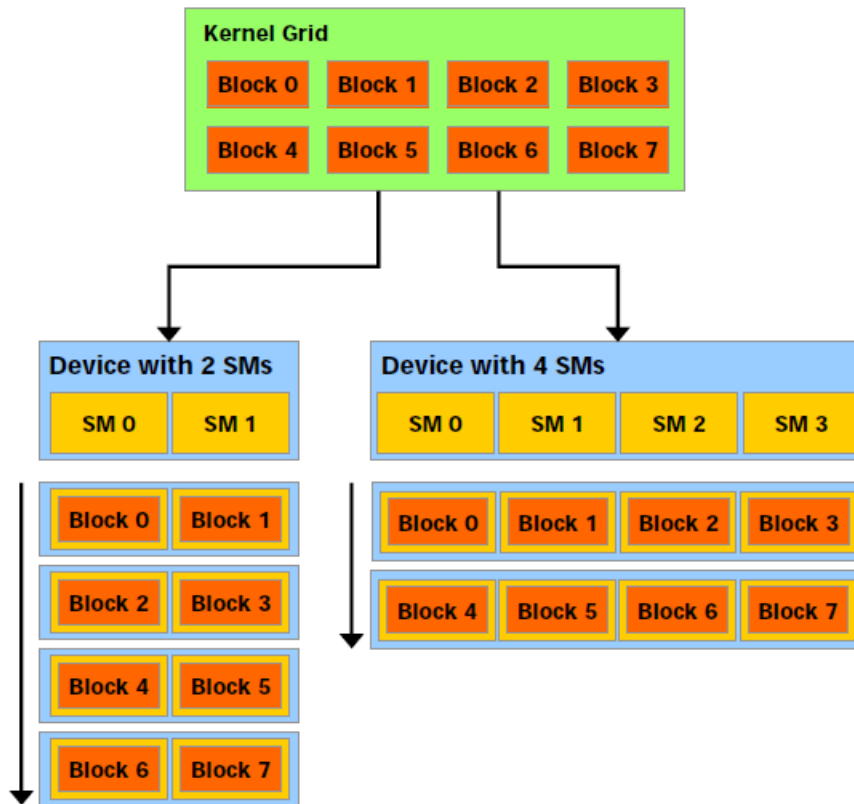


FIG. 3.2 – Répartition des blocs (figure provenant de la documentation NVIDIA)

3.2 La mémoire

3.2.1 La mémoire globale

Cette mémoire se trouve au niveau de la device memory sur le schéma 3.1. Elle est utilisée pour stocker les données en provenance du CPU et les données qui devront être renvoyées sur le CPU. Elle peut aussi servir pour les données qui doivent être accessibles par tous les threads. Étant en dehors des multiprocesseurs, l'accès à cette mémoire est très lent. C'est pourquoi de nombreux mécanismes ont été mis en place pour optimiser ces accès.

En plus de mémoires plus rapides, citées par la suite, des optimisations sont effectuées lorsque les threads accèdent à des zones mémoires contiguës et bien alignées. C'est ce qui est appelé de la mémoire coalescé. Ainsi quand la mémoire est correctement coalescé, 16 threads peuvent bénéficier d'un seul accès mémoire qui est ensuite redécoupé et distribué entre les 16 threads (un demi warp). Ce mécanisme est expliqué plus en détail sur le schéma 3.3. Le schéma explique le mécanisme avec 32 threads car on peut y retrouver l'architecture FERMI (compute capability 2.0) qui optimise les transferts sur des groupes de 32 threads. Les architectures précédentes n'optimisent que les transactions sur 16 threads. C'est pour ça que sur le dernier exemple on est obligé de faire une transaction de 92 octets à l'adresse 192. Même si cette partie de la mémoire a déjà été rapatriée par la transaction précédente, cela n'a été effectué que pour les 16 premiers threads. C'est pourquoi il faut la rapatrier une seconde fois pour les 16 threads suivants.

3.2.2 Autre mémoire distante

En plus de la mémoire globale, il y a la mémoire constante. La mémoire constante est de la mémoire globale que l'on spécifie comme étant constante. Elle subit alors des optimisations dont la plus importante est l'utilisation d'un cache spécifique au niveau de chaque multiprocesseur comme on peut le voir sur le schéma 3.1.

Une autre mémoire semblable est la mémoire de texture. C'est une mémoire placée elle aussi dans la device memory et qui est en lecture seule. Cette mémoire est optimisée pour les textures et donc les tableaux 2 dimensions de flottant simple précision (inutilisables avec du double précision). Comme la mémoire constante, elle bénéficie d'un cache sur chaque multiprocesseur.

Enfin la dernière mémoire se trouvant au niveau de la device memory est la mémoire locale. Si elle est appelée mémoire locale alors qu'elle se trouve en dehors des multiprocesseurs, c'est à cause de sa visibilité. La mémoire locale est utilisée automatiquement lorsque les threads utilisent trop de registres. C'est à dire que le nombre de registres sur les multiprocesseurs ne sont pas suffisants

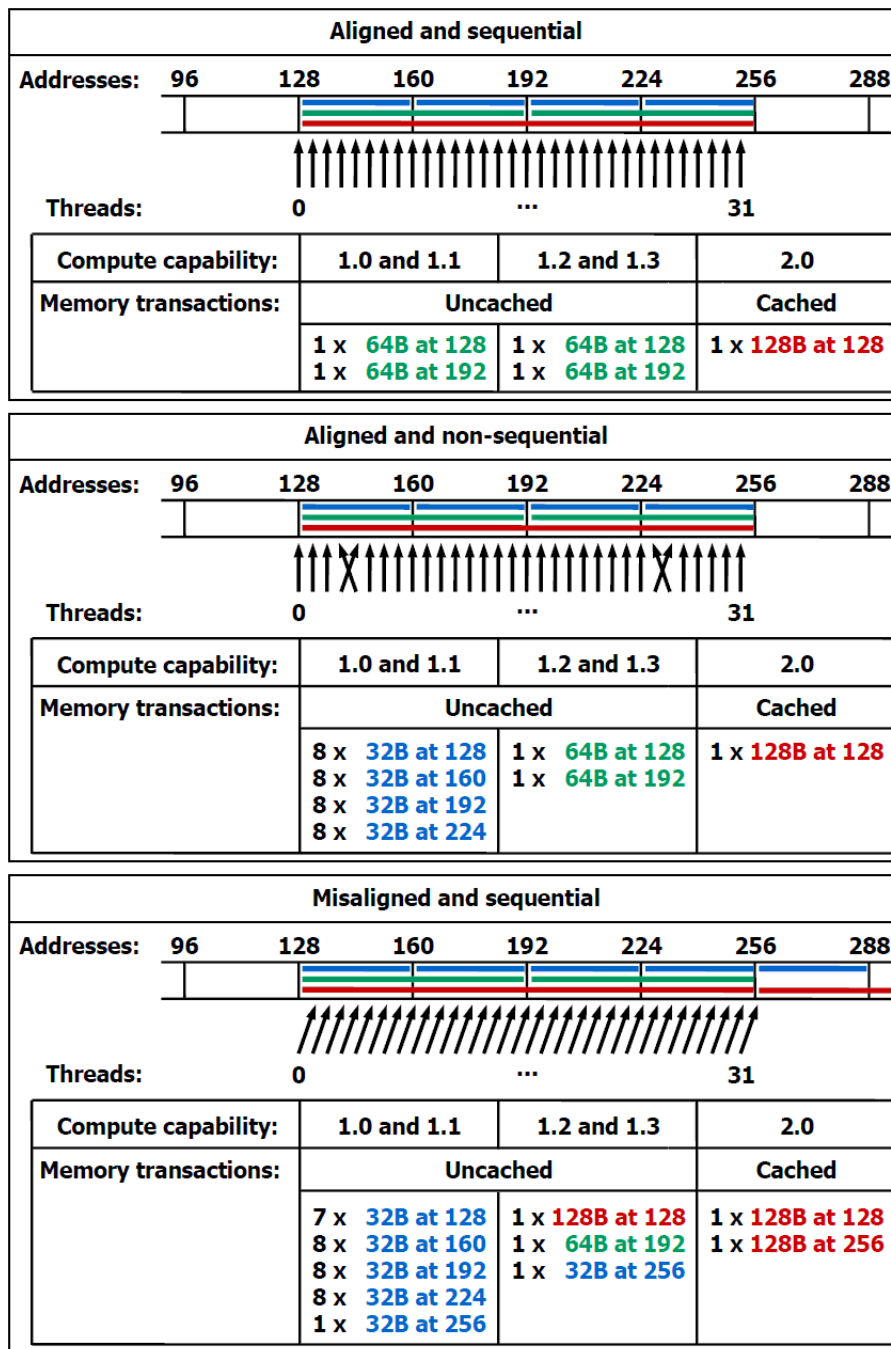


FIG. 3.3 – Optimisation des accès mémoire (figure provenant de la documentation NVIDIA)

ou que l'on a spécifié un nombre maximum de registres par thread insuffisant. Il est cependant conseillé d'éviter au maximum son utilisation, la mémoire se trouvant en dehors des multiprocesseurs, les accès en sont très coûteux et la performance du noyau de calcul chute grandement.

3.2.3 La mémoire sur les multiprocesseurs

Il reste maintenant les mémoires qui sont directement sur les multiprocesseurs. Ces mémoires sont plus rapides que les mémoires précédentes mais ont une visibilité réduite. Les deux caches pour la mémoire constante et la mémoire de texture ont déjà été abordés. Il y a ensuite la mémoire partagée. La mémoire partagée est une mémoire qui est partagée entre les threads d'un même bloc. Étant beaucoup plus rapide que la mémoire globale, elle est principalement utilisée pour optimiser les accès mémoire à la manière d'un cache. On peut aussi s'en servir pour coalesce les accès mémoire.

Pour la rendre la plus rapide possible, la mémoire partagée a été découpée en banks. La mémoire partagée est composée de 16 banks organisés de manière à ce que les mots de 32 octets successifs soient assignés à des banks successifs. L'avantage des banks est que chaque bank peut être accédé indépendamment des autres. Comme pour la mémoire globale, les accès mémoire sont optimisés par demi-warp (16 threads). Ainsi, si chaque thread du demi-warp accède à la mémoire d'un bank différent, les accès mémoires sont effectués en parallèle. Par contre, si un bank est accédé par plusieurs threads, ce qu'on appelle "conflit de bank", les accès à ce bank seront séquentialisés. Sauf si tous les threads du demi-warp accèdent à la même adresse mémoire, dans ce cas un broadcast est effectué par la mémoire qui se traduit par un seul accès mémoire pour tous les threads du demi-warp.

3.2.4 Classement des mémoire

Il y a deux manières de classer les différentes mémoires des GPU. On peut soit les classer par visibilité, ce que l'on retrouve sur le schéma 3.4. Soit les classer par rapidité d'accès. Autant le classement des mémoires par visibilité est clairement défini par NVidia, autant la vitesse de chaque mémoire n'est pas toujours bien renseignée. Il y a, par contre, certains articles qui se sont penchés sur la question et qui ont mesuré les différents temps d'accès des différentes mémoires ([4], [5]). Au final le classement des mémoires du plus lent au plus rapide est le suivant :

1. mémoire globale / mémoire locale (400-600 cycles),
2. mémoire constante / mémoire de texture (dépendant du cas),
3. mémoire partagée (35 cycles),
4. registres.

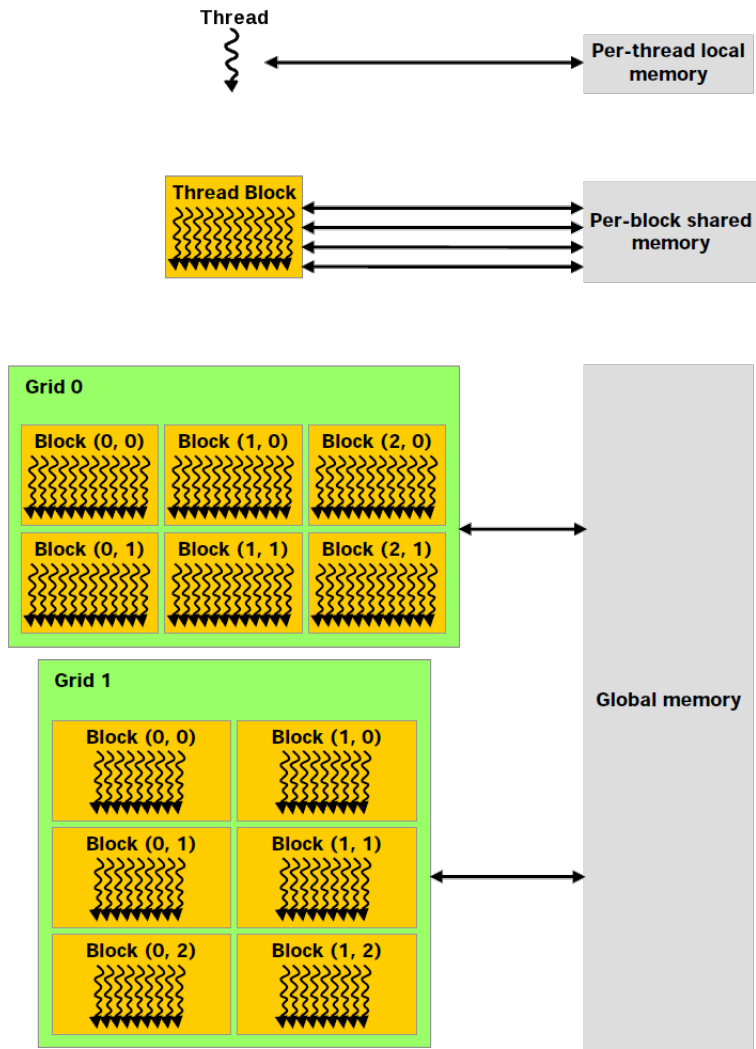


FIG. 3.4 – Visibilité de la mémoire (figure provenant de la documentation NVIDIA)

3.3 FERMI

FERMI est la nouvelle architecture des cartes graphiques NVidia. Cette architecture a été conçue en prenant en compte le retour des développeurs CUDA et des utilisateurs des GPU pour le calcul scientifique. Elle offre de nombreuses nouvelles fonctionnalités ([2]) dont les principales sont :

- ★ Troisième génération de Multiprocesseur Streaming
 - 32 coeurs par SM (4x plus)
 - 8x le pic de performance pour le calcul double précision
 - 2 ordonnanceurs par SM permettant l'ordonnancement de 2 warps indépendants simultanément
 - 64 KB de RAM avec 2 types de partitionnement entre un cache L1 et la mémoire partagée
- ★ Seconde génération de threads parallèles
 - espace d'adresse unifié avec support c++ complet
 - optimisé pour OpenCL and DirectCompute
 - respect de la norme IEEE 754-2008 pour simple et double précision
 - support adressage 64-bit
 - amélioration des performances grâce à la prédiction
- ★ Subsysteme mémoire amélioré
 - Mémoire partagée de 32 banks
 - Accès mémoire par warp et non demi-warp pour optimisation (mémoire coalesce, conflit de bank)
 - NVIDIA Parallel DataCache™ hierarchy avec cache L1 configurable et cache L2 unifié
 - support ECC (Error Correcting Codes)
 - Amélioration des performances des opérations atomiques
- ★ NVIDIA GigaThread™ Engine
 - changement de contexte 10x plus rapide
 - exécution concurrent des noyaux de calcul
 - Recouvrement des transferts bi-directionnels

Chapitre 4

Les langages de développement

Il y a plusieurs solutions pour développer une application sur GPU. Ces solutions ont toutes des avantages et inconvénients différents : la maintenabilité, la portabilité, les performances... Le but du stage étant d'obtenir des performances, il a été décidé de faire le portage en CUDA C (le langage développé par NVIDIA). Mais j'ai testé d'autres langages en début de stage et effectué quelques comparaisons pour la prise de décision. Les différents langages testés vont être présentés par la suite. Pour chaque langage, je présenterais la traduction de l'exemple de code Fortran ci-dessous. Cela permettra de se faire une idée du langage, de sa complexité et de la rapidité de portage du code.

```
0  subroutine cpuCompute (n, a, r)
C_IN
    integer :: n
    real    :: a(n)
C_OUT
5  real    :: r(n)
C_LOC
    real :: s, c
    integer :: i

10     DO i = 1, n
        s = sin(a(i))
        c = cos(a(i))
        r(i) = s*s + c*c
15     END DO

    END
```

4.1 CUDA C

Le CUDA C est un langage permettant de développer des applications pour les GPU NVIDIA. Le CUDA C n'est, en fait, qu'une extension du C/C++ ajoutant un certain nombre de fonctions et de mots clés permettant de manipuler

un GPU. Le GPU supporte entièrement le C mais ne supporte qu'une partie limitée du C++. Par contre le support du C++ évolue et augmente à chaque nouvelle version du driver CUDA. Le langage a été créé et est maintenu par NVIDIA. Cela assure qu'il évolue avec les GPU NVIDIA et qu'il soit proche de l'architecture ce qui permet d'obtenir les meilleures performances possibles. Il y a deux façons de développer en CUDA C : soit en passant par la CUDA RUNTIME LIBRARY ou en utilisant le CUDA Driver API. Au début du stage, il fallait obligatoirement choisir l'un ou l'autre. Mais depuis le driver CUDA 3.0, une application peut utiliser les deux solutions en même temps.

4.1.1 CUDA RUNTIME LIBRARY

La librairie runtime de CUDA ajoute un nombre limité de fonctions et de mots clés au langage C. Et de nombreuses opérations sont faites implicitement comme la création de contexte, le choix du GPU... Cela permet une prise en main beaucoup plus simple et rapide du CUDA. Ci-dessous notre exemple de code traduit en CUDA C RUNTIME :

```

0  __global__ void gpuCompute(int n, float * d_a, float * d_r )
   {
   // Compute thread Id
   int idx = threadIdx.x + blockIdx.x*blockDim.x ;
   float s,c,a;

5   if (idx<n) {
       a=d_a[idx];
       s = sinf(a);
       c = cosf(a);

10      d_r[idx] = s*s + c*c;
   }
   }

0  // allocate device memory
   cudaMalloc( (void **) &d_a, n*sizeof(float) );
   cudaMalloc( (void **) &d_r, n*sizeof(float) );

   // Copy host array to device array
5  cudaMemcpy( d_a, a, n*sizeof(float), cudaMemcpyHostToDevice );

   // Invoke kernel
   dim3 dimGrid(numBlocks,1,1);
   dim3 dimBlock(numThreadsPerBlock,1,1);
10  gpuCompute<<< dimGrid, dimBlock>>>( n, d_a, d_r );

   // Copy device result array to host result array
   cudaMemcpy( r, d_r, n*sizeof(float), cudaMemcpyDeviceToHost );

15  // Free device memory
   cudaFree(d_a);
   cudaFree(d_r);

```

Le premier code correspond au noyau de calcul qui sera exécuté sur GPU. Le mot clé `__global__` devant le nom de la fonction spécifie que la fonction sera exécutée sur GPU mais appelée à partir du CPU. On a ensuite en ligne 3, le calcul de l'index du thread : indice du thread dans le bloc actuel + indice du bloc * taille du bloc. On n'utilise dans cette exemple que l'axe x pour la

géométrie d'exécution, l'utilisation des autres axe étant similaire. Puis le calcul est effectué. Le deuxième code correspond à l'appel du noyau de calcul. Il sera intégré à notre code CPU à l'endroit où l'on veut effectuer le calcul. L'appel au noyau de calcul se fait à la ligne 10. Au tout début (lignes 0-5), la mémoire sur GPU est allouée et les données y sont copiées. Ensuite au niveau des lignes 8 et 9, la géométrie d'exécution du noyau est définie. Enfin après l'appel du noyau, le résultat est rapatrié sur CPU et la mémoire GPU est libérée.

L'exemple regroupe les principales fonctions utilisées de la Runtime library. Il y a d'autres mécanismes assez importants, surtout pour l'optimisation. Il y a l'utilisation des mémoires spéciales vu dans la partie 3.2 (shared memory, constant memory...).

L'allocation de la mémoire PIN au niveau du CPU. La mémoire PIN est de la mémoire verrouillée qui ne sera pas swappée. Elle permet d'accélérer les transferts mémoire, de faire des transferts Asynchrones ou de faire du Zero-Copy. Le ZeroCopy consiste à accéder directement à la mémoire CPU à partir d'un noyau de calcul GPU. Il y a, par contre, des inconvénients à l'utilisation de la PIN memory. Premièrement elle est limitée par le système, cela évite de pouvoir allouer trop de PIN memory au risque de tout allouer et de bloquer la machine. De plus elle décroît les performances CPU. Cette mémoire ne pouvant être swappée, l'espace mémoire restant sera réduit et sera swappé plus souvent.

Il y a aussi des fonctions de manipulation de tableau 2d (allocation, transfert...). On a vu dans la partie 3.2, que les accès mémoires étaient optimisés quand les accès étaient contiguës et alignés. Lorsque l'on a un tableau 2d, le début de chaque ligne (ou de chaque colonne) n'est pas forcément bien aligné (des lignes de $10 \times 16 + 5$ float par exemple). Les fonctions spécifiques aux tableaux 2D alloueront ainsi 11×16 floats par lignes. Certaines cases seront inutiles, et les transferts mémoires plus importants. Mais chaque début de ligne sera bien aligné et au final les performances seront meilleures.

4.1.2 CUDA Driver API

L'API CUDA est bien plus complète que la librairie runtime. Elle offre une plus grande liberté d'action. Toutes les actions peuvent être faites explicitement contrairement à la librairie runtime. On a plus de contrôle sur ce que l'on fait, plus particulièrement lorsque l'on a plusieurs GPU on choisit directement celui que l'on utilise. Par contre la prise en main est plus difficile et la programmation est plus longue. Ci-dessous, la traduction du code de base avec l'API driver :

```
0  __global__ void gpuCompute(int n, float * d_a, float * d_r )
   {
   // Compute thread Id
   int idx = threadIdx.x + blockIdx.x*blockDim.x ;
   float s,c,a;
5
   if (idx<n) {
       a=d_a[idx];
```

```

    s = sinf(a);
    c = cosf(a);
10     d_r[idx] = s*s + c*c;
    }
}

0 // Get handle for device 0
CUdevice cuDevice;
cuDeviceGet(&cuDevice, 0);

// Create context
5 CUcontext cuContext;
cuCtxCreate(&cuContext, 0, cuDevice);

// Create module from binary file
CUmodule cuModule;
10 cuModuleLoad(&cuModule, "GPUmodule.ptx");

// allocate device memory
int size = n*sizeof(float);
CUdeviceptr d_a;
15 cuMemAlloc(&d_a, size);
CUdeviceptr d_r;
cuMemAlloc(&d_r, size);

// Copy host array to device array
20 cuMemcpyHtoD(d_a, a, size);

// Get function handle from module
CUfunction gpuCompute;
cuModuleGetFunction(&gpuCompute, cuModule, "GPUmodule");

25 // Configuration kernel parameters
#define ALIGN_UP(offset, alignement) \
    ((offset) + (alignement) - 1) & ~((alignement) - 1)
int offset = 0;
void* ptr;
30 ptr = (void*)(size_t)d_a;
ALIGN_UP(offset, _alignof(ptr));
cuParamSetv(gpuCompute, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
35 ptr = (void*)(size_t)d_r;
ALIGN_UP(offset, _alignof(ptr));
cuParamSetv(gpuCompute, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
cuParamSetSize(gpuCompute, offset);

40 // Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
cuFuncSetBlockShape(gpuCompute, threadsPerBlock, 1, 1);
45 cuLaunchGrid(gpuCompute, blocksPerGrid, 1);

// Copy device result array to host result array
cuMemcpyDtoH(r, d_r, size);

50 // Free device memory
cuMemFree(d_a);
cuMemFree(d_r);

// Exit CUDA
55 cudaThreadExit();

```

Premièrement on peut remarquer que le noyau de calcul reste inchangé. En effet, il n'y a aucune différence dans les noyaux CUDA entre les deux bibliothèques. Par contre, il y a une différence flagrante dans l'appel du noyau. Ici

le code pour appeler le noyau est bien plus important. En réalité, le code pour appeler le noyau est le code de la ligne 12 à la ligne 52. Avant la ligne 12, on configure le processus pour utiliser la première carte. Après la ligne 52 on libère toute la configuration du GPU pour le processus.

Commençons par examiner le code de la ligne 12 à 52. La première différence est les pointeurs utilisés pour les données GPU. On utilise ici un type spécial : *CUdeviceptr*. Ensuite il y a l'allocation, les transferts et la libération. Ces opérations sont semblables à celles de la bibliothèque runtime. Seul le nom et le prototype des fonctions sont modifiés. La deuxième différence importante visible est l'utilisation de module. Les noyaux de calcul CUDA sont contenus dans des modules compilés à part du code C ou C++ puis importés (ligne 24). Enfin l'appel de fonction diffère grandement. Il faut empiler les paramètres de l'appel du noyau (lignes 26-39) puis configurer la géométrie du noyau (lignes 42-44) pour enfin lancer l'exécution du noyau sur GPU (ligne 45).

Le contexte CUDA est souvent comparé aux processus CPU. Un contexte CUDA encapsule toutes les ressources et les actions effectuées (dans le cadre du contexte) relatif à CUDA. Chaque thread/processus CPU utilisant un GPU a un et un seul contexte CUDA à un instant t . Mais on a la possibilité d'empiler les contextes. Dans ce cas, le dernier est le contexte courant. Lorsque l'on dépile ou que l'on détruit le contexte courant, celui qui est juste en dessous dans la pile devient le contexte courant. On a aussi la possibilité de détacher ou d'attacher un contexte à un processus. Cela permet par exemple de partager un contexte entre plusieurs processus en le détachant d'un processus pour l'attacher à un autre. On ne peut attacher un contexte que s'il n'est pas déjà attaché à un processus. Enfin si aucun contexte n'existe lors d'un appel à une fonction CUDA utilisant le GPU alors un contexte est créé automatiquement.

4.2 CUDA Fortran

Le CUDA Fortran est basé sur le même principe que le CUDA C. Contrairement au CUDA C, le CUDA Fortran n'est pas développé par NVidia mais par PGI. NVidia a pris la décision de se concentrer sur les GPU et le CUDA C et de ne pas développer de version Fortran de CUDA. Mais une version CUDA Fortran est un plus important pour leurs GPU. C'est pourquoi, NVidia travaille en coopération avec PGI sur le développement du CUDA Fortran. Cela signifie aussi que contrairement au CUDA C qui est gratuit, le CUDA Fortran est sous licence PGI.

Au niveau du langage, le CUDA Fortran est comparable au CUDA C. On retrouve les mêmes fonctionnalités. La plupart du temps, seul le nom des fonctions diffère entre les fonctions CUDA C et les fonctions CUDA Fortran. De temps en temps, par contre, les fonctions sont un peu modifiées pour être plus adaptées à l'esprit Fortran. Examinons l'exemple porté en CUDA Fortran :

```

0  MODULE gpuComputeMod
   USE cudafor
   CONTAINS

   ATTRIBUTES(GLOBAL) SUBROUTINE gpuCompute (n, d_a, d_r)
5  C_IN
   integer , value :: n
   real      :: a(n)
   C_OUT
   real      :: r(n)
10 C_LOC
   real :: s, c, a
   integer :: i

   C_Compute thread Id
15   i = (blockidx%x-1) * blockdim%x + threadidx%x

   IF (i.LE.n) THEN
     a=d_a(i)
     s = sin(a)
20     c = cos(a)

     d_r(i) = s*s + c*c
   END if

25   END SUBROUTINE gpuCompute
   END MODULE gpuComputeMod

```

```

0  use cudafor
   use gpuComputeMod

   real*8, device, allocatable, dimension(:) :: d_a, d_r
   type(dim3) :: dimGrid, dimBlock
5
   (...)

   C_allocate device memory
   allocate( d_a(n), d_r(n))
10
   C_Copy vectors from host memory to device memory
   d_a=a

   C_Invoke kernel
15   dimGrid = dim3( numBlocks, 1, 1 )
   dimBlock = dim3( numThreadsPerBlock, nfld, 1 )
   call gpuCompute<<<dimGrid,dimBlock>>>(n, d_a, d_r)

   C_Copy device result array to host result array
20   r=d_r

   C_Free device memory
   deallocate(d_a, r_a)

```

On retrouve dans le noyau en CUDA Fortran la même structure qu'en CUDA C. On calcule l'index du thread et on remplace la boucle par un if. Il faut, par contre, faire attention, le noyau de calcul est placé dans un module. Ici le noyau est placé dans le module `gpuComputeMod`. Ceci est obligatoire. Si le noyau n'est pas placé dans un module Fortran, il y aura un problème à la compilation lors du linkage.

Ensuite pour l'appel du noyau on a aussi une structure similaire au CUDA C. On retrouve les 5 étapes : allocation de la mémoire GPU, transfert des données sur le GPU, exécution du noyau, transfert des données sur le CPU et libération

de la mémoire. Mais il faut faire attention au transfert. Contrairement au CUDA C où on devait utiliser une fonction bien explicite, ici un signe égal suffit. Mais le temps de transfert est toujours le même, il faut bien avoir conscience que ce signe effectue un transfert dont le temps n'est pas négligeable. Cependant, il existe en CUDA Fortran des fonctions similaires aux fonctions de transfert CUDA C. On a ainsi le choix de les utiliser si on veut éviter d'utiliser simplement le signe égal et risquer de faire trop souvent des transferts. Enfin, même si sur l'exemple le CUDA Fortran semble plus proche du CUDA runtime library, le CUDA Fortran possède des fonctions permettant de choisir les GPU, de manipuler les context GPU...

4.3 Autres

Il y a d'autres solutions pour développer sur GPU. Une autre solution, assez connue, est l'OpenCL (Open Computing Language). L'OpenCL est le premier standard open source permettant le développement d'applications pour architecture parallèle. Dans cette optique, il permet de créer des applications fonctionnant sur les GPU NVidia ou ATI mais aussi sur d'autres architectures parallèles. Il a donc l'avantage d'être portable et de ne pas être soumis à une architecture trop spécifique. Cependant l'OpenCL est récent et n'est pas assez spécifique à l'architecture NVidia. C'est pourquoi il obtient de moins bonnes performances que le CUDA C. Enfin, même s'il est basé sur le même principe que le CUDA (ajout de fonction et mot clé au langage C), il est plus difficile à prendre en main. Au final, je n'ai pas testé l'OpenCL à cause des performances moindres pour un développement non facilité et une portabilité n'étant pas une priorité pour ce stage.

Par contre, lors de mes tests, j'ai aussi testé deux solutions par directives : PGI et HMPP. Ces deux solutions permettent de placer des directives de compilations dans du code C/C++ ou Fortran. La solution offerte par PGI permet de définir des régions de code à exécuter sur GPU et de gérer les transferts mémoire. Elle offre aussi certaines directives pour l'optimisation du noyau généré. Mais cela reste très succinct. HMPP permet la même chose. Sauf que PGI est spécifique à CUDA alors que HMPP permet de générer aussi du code Brook+ pour les cartes ATI, de l'OpenCL... Il offre aussi plus de directives permettant d'optimiser les noyaux de calcul générés. Cependant, le principal avantage des directives est la portabilité. En effet, si on ne veut pas utiliser de GPU il suffit d'ignorer les directives lors de la compilation. Or, l'architecture des GPU, comme on l'a déjà vu, est très particulière. Ainsi si on veut vraiment optimiser un noyau de calcul il va falloir modifier le code pour avoir de la mémoire alignée, utiliser la mémoire partagée... Au final il faudra modifier le code. A partir de là on perd l'avantage de la portabilité. HMPP permet de spécifier des régions de code qui seront prises en compte en fonction de la génération demandée. Cela permet d'avoir une partie de code pour le CUDA, une partie pour l'OpenCL et une partie pour la version CPU de base. Mais dans ce cas là, on se

retrouve avec 3 versions du code à maintenir.

Chapitre 5

Le projet elsA

5.1 Présentation

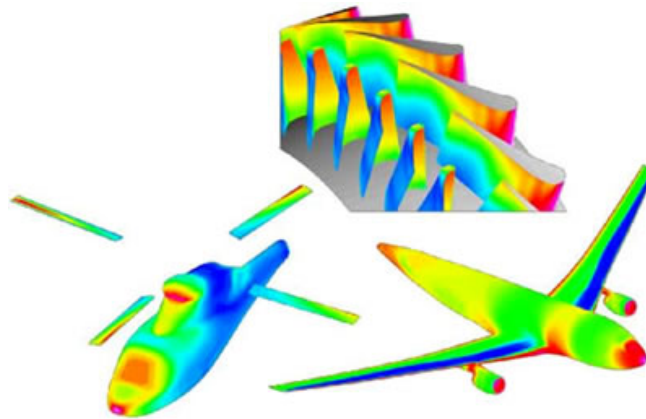


FIG. 5.1 – Simulations d'écoulements turbulents en aérodynamique externe et interne

Le projet elsA, Ensemble Logiciel pour la Simulation en Aérodynamique, est un code de simulation numérique regroupant diverses applications possibles du domaine de l'aérodynamique. On y retrouve les disciplines suivantes :

- aérodynamique, aéroélasticité, couplage aérothermique, couplage aéro-acoustique ;
- avion, hélicoptère, turbo machine, missiles, lanceurs, admission d'air, buse d'éjection, réacteur à propulsion ;
- recherche et application industrielle ;
- simulation Euler, RANS, URANS, DES, LES ;

- hypothèse des gaz parfaits mono espèce avec une valeur spécifique du rapport de chaleur choisie par l'utilisateur
- calcul de la sensibilité pour une conception optimale.

Le projet fut fondé en 1997 par l'ONERA, ce n'est qu'en 2001 que le CERFACS a rejoint le projet. L'idée, à l'origine du projet, était de rassembler toutes les connaissances et savoir-faire en mécanique des fluides (CFD) dans un seul logiciel. Cela permettrait d'avoir un outil polyvalent avec lequel on peut choisir les différents calculs que l'on souhaite effectuer pour simuler n'importe quelle situation d'aérodynamique. Au final, elsA est une application très structurée et modulaire pour permettre justement l'ajout de nouveaux modules correspondant à de nouveaux calculs et de choisir les différents calculs effectués sur notre maillage.

5.2 Structure du code

Comme il est dit dans la partie précédente, elsA est très bien structurée et modulaire. Tous les calculs sont répartis dans des noyaux de calcul bien séparés les uns des autres et codés en Fortran. Au dessus de ces noyaux de calcul, il y a une architecture orientée objet codée en C++. Les classes C++ permettent de gérer les noyaux de calcul, les maillages et l'environnement de la simulation. Enfin l'interface utilisateur est faite en python. Un utilisateur doit construire des scripts python initialisant différentes classes de la partie C++ correspondant à la description du problème, des calculs à effectuer...

Le code d'elsA est regroupé en divers modules. Les modules sont construits en suivant un découpage fonctionnel du code. On a ainsi les groupes de modules suivants :

- factory : modules générant les différents objets
- solver : modules construisant et gérant les équations
- space discretization : modules responsables des conditions limites et des termes des équations
- physical model : modules contenant les modèles physiques
- geometry : modules gérant la géométrie et topologie des problèmes
- base : modules de bas niveau (stockage des données, gestion de la parallélisation)

5.3 Compilation

La compilation se fait en 3 temps. Dans un premier temps on génère les makefile pour chaque module, dans un deuxième temps on compile tous les modules puis on effectue l'édition des liens. Cette compilation prend du temps. Bien sur, si on ne modifie que le code, il n'est pas nécessaire de régénérer les makefile, sauf si on modifie les fichiers d'en-tête (file.h). De même, si on compile dynamiquement, l'édition des liens n'est pas à refaire à chaque fois que l'on modifie le code.

La première chose à faire lors de la compilation est de choisir la version de production. La génération de makefile se fait à partir d'autres makefile. Ainsi il y a une liste de makefile correspondant chacun à une version de production différente. Par exemple, il y a la version intelx86 ou PGI10... A ce nom de version on ajoute des mots clés tels que `_so`, `_r8`. Cela permet de définir des options de compilation. Par exemple `_so` active une compilation dynamique, `_r8` permet l'utilisation de la double précision. Le précédent stagiaire avait créé une version propre à l'utilisation du GPU. Cependant on peut vouloir utiliser les compilateurs INTEL ou PGI avec la version GPU, il m'a semblé plus approprié de créer le mot clé `_gpu` pour activer la compilation de la version GPU. Ensuite on lance la génération des makefile qui produit un makefile dans chaque dossier du code correspondant aux différents modules.

Dans chaque dossier de module on peut trouver un fichier `Make_obj`. Ce fichier contenait la déclaration d'une variable, contenant tous les objets à générer, qui était recopiée dans le makefile généré précédemment. Si on ajoute les fichiers CUDA dans cette variable, comme c'était le cas au départ, cela permet de compiler la version GPU. Cependant cela générerait une erreur pour la compilation de la version CPU puisque la version CPU ne connaît pas les fichiers CUDA. Utiliser un `#ifdef` n'a pas fonctionné non plus car lors de la génération du makefile de la version CPU il générerait des espaces à la place des `#ifdef`. Ces espaces rendaient le makefile généré non fonctionnel. Au final j'ai modifié les `Make_obj` pour générer une autre liste de variables spécifiques au GPU qui est ajoutée à la première variable si on veut compiler la version GPU.

Chapitre 6

Le portage

Le développement sur GPU étant relativement nouveau, aucun planning n'a été fait en début de stage. J'ai commencé avec le but d'accélérer un cas test en portant des routines sur GPU. Tous les mois, une réunion, regroupant les personnes concernées par le portage, avait lieu. Lors de cette réunion, on étudiait l'avancement du portage et les résultats obtenus. A l'issue de ces réunions, les nouvelles tâches à effectuer étaient choisies. On peut voir les différentes tâches effectuées lors du stage sur le diagramme de gant de la figure 6.1.

Le cas test est un calcul turbulent instationnaire dans un canal. Il m'a été fourni avec la possibilité de créer des maillages. J'ai ainsi pu générer 6 maillages de tailles différentes pour avoir un échantillon représentatif des tailles de maillage utilisées habituellement avec elsA. Les tailles des maillages sont les suivantes :

- 65x129x61 (511485 points)
- 65x129x161 (1349985 points)
- 65x229x161 (2396485 points)
- 65x229x261 (3884985 points)
- 65x329x261 (5581485 points)
- 65x329x361 (7719985 points)

6.1 Portage des routines

6.1.1 Première stratégie de portage

Dans un premier temps divers profilings du cas test furent réalisés avec gprof et pgprof pour confirmer les résultats. Les profilings ont été effectués sur des tailles et des nombres d'itérations différents pour vérifier si cela avait un impacte sur les routines. Il s'est avéré que tous les profilings sont équivalents. Le profiling de la figure 6.2 est un profiling effectué sur le maillage de taille 65x329x361 pour 200 itérations. On remarque que 65.83% du temps d'exécution est compris dans 5 routines seulement. Ce qui est un bon résultat pour l'optimisation du cas test.

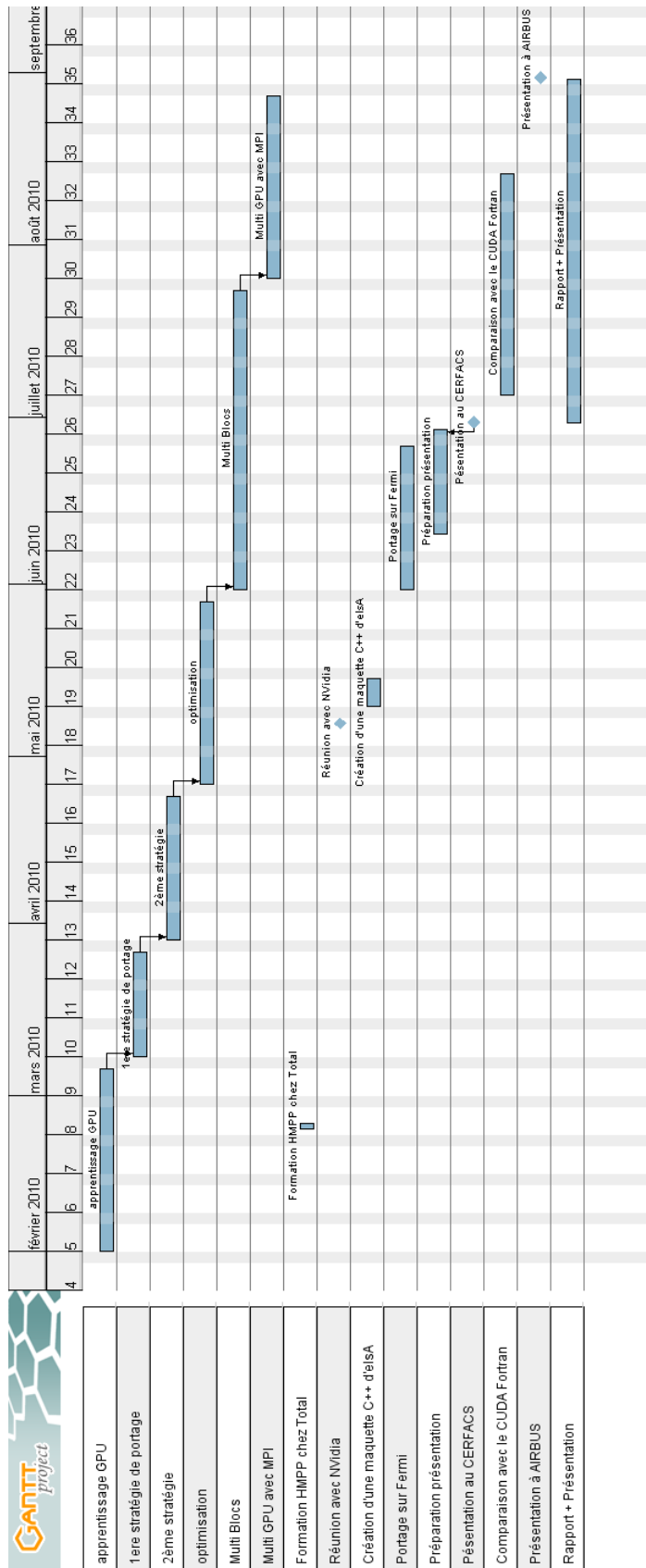


FIG. 6.1 – Diagramme de Gant du stage

A la suite de ces profilings, une première stratégie de portage fut choisie. Cette dernière consistait à porter les 3/4 premières routines du profiling. Stratégie la plus utilisée lors des optimisations, puisque pour une accélération équivalente d'une routine, le gain global sera meilleur si la routine fait partie du haut du profiling que si elle fait partie du bas. La routine *compfluxintgradcencorrprim* avait déjà été portée sur GPU à la suite du précédent stage à l'ONERA (voir rapport [3]). La première action fut de vérifier le portage de cette routine pour voir comment cela avait été fait et s'il n'y avait pas d'amélioration possible. Ensuite les routines *fxcroe5* et *fxccomp3o* furent portées sur GPU. Ces deux routines sont très proches dans le code, elles ne sont séparées que par une troisième routine (*complrstate*). Dans le but de minimiser les transferts entre CPU et GPU et ainsi augmenter le gain du portage, j'ai décidé de porter cette troisième routine. Dans la même optique de minimisation des transferts, c'est à ce moment là, que j'ai décidé de faire quelques classes C++ pour gérer les transferts. Ce système de gestion des transferts est détaillé plus loin dans la partie 6.3.

6.1.2 Deuxième stratégie de portage

Une fois ces routines portées d'autres profilings furent effectués avec *cudaprof* (outil de NVIDIA) pour analyser les résultats et trouver où étaient les optimisations à effectuer. Ce profiling est présent sur la figure 6.3. Le facteur limitant étant clairement les transferts entre le CPU et le GPU (>60%), j'ai décidé de réduire ces transferts. Pour cela, j'ai choisi d'éliminer le rapatriement sur CPU d'un certain tableau de données nommé *fxd*. Ce tableau fut choisi pour sa taille (3 fois plus gros que les autres) et pour le sens du transfert. En effet les transferts sont plus lent du GPU vers le CPU que dans l'autre sens.

Pour éliminer ce transfert, il a fallu porter sur GPU toutes les fonctions utilisant ou modifiant *fxd*. Ces fonctions furent trouvées en utilisant un débogueur et en suivant le tableau *fxd*. Au final, 3 autres fonctions furent portées sur GPU pour éliminer le transfert : *compintfluxgradcorrbdminusone*, *fxdcp1borextrap* et *compbalancevec3*. De plus, le transfert ne fut pas tout à fait éliminé, *fxd* est utilisé pour calculer un autre tableau 3 fois plus petit. Le transfert fut remplacé par un autre transfert mais plus petit et plus rapide. Le problème est qu'il a fallu porter d'autres routines sur GPU. Pour pouvoir effectuer les calculs de ces trois routines, il a fallu transférer plus de données sur GPU. Donc même si le temps de transfert du GPU vers le CPU fut réduit, celui du transfert du CPU vers le GPU fut augmenté. Malgré tout, l'exécution final fut plus rapide, mais l'accélération fut très faible (gain $\approx 5\%$)

compintfluxgradcorrbdminusone et *fxdcp1borextrap* sont les deux fonctions qui ont augmenté considérablement le nombre de transferts du CPU vers le GPU. De plus ce sont des fonctions courtes et assez rapides. Le gain du portage sur GPU n'est pas très significatif. Enfin ces deux fonctions servent à calculer un tiers de *fxd*. Calcul qui peut être effectué sur le CPU sans avoir à rapatrier

Profiling CPU Taille 6 Iter 200					
%	cumulative seconds	self			name
		seconds	calls	ks/call	
24.6	2416.33	2416.33	2400	0	fxcroe5_
21.94	4571.58	2155.25	800	0	compfluxintgradcencorprim_
7.23	5282.08	710.5	2400	0	fxccomp3o_
6.12	5883.61	601.53	3207	0	compgradsca2_
5.94	6467.53	583.91	1600	0	compbalancevec3_
4.67	6926.62	459.09	2402	0	cons2speedf_
4.53	7371.8	445.18	9604	0	cons2tempf_
3.73	7738.25	366.45	1603	0	operatordivf_
3.54	8085.96	347.71	800	0	cons2prim_
2.14	8296.49	210.53	2400	0	fdaddpart_
1.78	8471.14	174.65	800	0	TmoSolverBase::computeLhs()
1.54	8621.98	150.84	800	0	cons2primts_
1.08	8727.98	106	800	0	lhsupdate_
0.93	8819.34	91.36	6400	0	compgradborder_
0.87	8905.06	85.72	1600	0	compintfluxgradcorrbdminusone_
0.86	8989.52	84.46	2401	0	compmu_
0.72	9060.47	70.95	800	0	FldFieldF::mulzca_mul_assign
0.71	9129.86	69.39	201	0	cpdtconv_
0.63	9191.99	62.13	6414	0	compbndgrad_
0.6	9251.09	59.1	201	0	turcompmutwale_
0.44	9294.64	43.55	800	0	SouForcTerm::compute
0.44	9337.66	43.02	6406	0	extrap0f_
0.42	9378.8	41.14	200	0	TmoSolverBase::solutionPEqualSolutionN
0.42	9419.84	41.04			spec_dexp2
0.4	9458.91	39.08	4863649413	0	FldFieldF::operator[](int)
0.35	9493.12	34.21	800	0	FldFieldF::add_and_assign
0.32	9524.39	31.27	2002	0	isothwallg_
0.31	9555.2	30.81	800	0	FldFieldF::operator/=
0.31	9585.27	30.07	1004	0	setallvaluesatf_
0.28	9612.84	27.57	802	0	FldFieldF::mulzca_and_assign
0.27	9639.02	26.18	568682145	0	FldArrayF::operator[](int)
0.26	9664.97	25.95	1600	0	fxdcp1borextrap_
0.26	9690.03	25.06	201	0	cptimesteptur_
0.24	9713.19	23.16	1600	0	complrstate_
0.14	9727.33	14.14	800	0	FldFieldF::mulzca_add_assign
					(...)

FIG. 6.2 – Profiling CPU du cas test

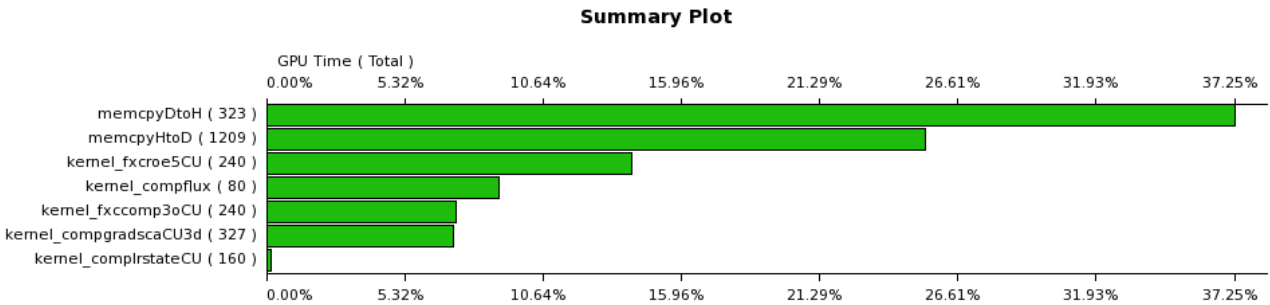


FIG. 6.3 – Profiling d'elsA GPU version 1.0

de donnée du GPU. J'ai essayé de laisser le calcul des fonctions sur CPU et de ne transférer que le résultat pour ensuite le recopier dans le fxd sur GPU. Le gain du portage des deux fonctions est perdu mais 11 transferts sont remplacés par seulement 2 transferts et 2 copies sur le GPU. Au final, cela accélère très légèrement les calculs.

6.1.3 Optimisation

Accès Mémoire

La principale source d'optimisation se trouve, en général, dans l'optimisation des accès mémoire. Les accès mémoire sont beaucoup plus lents que le calcul. Il faut minimiser le temps des accès mémoire. Il y a plusieurs méthodes pour cela :

1. éliminer les répétitions des transferts en utilisant les registres ou en remaniant le calcul,
2. coalescer les accès mémoire pour permettre le rapatriement des données de 8 threads en un seul transfert,
3. utiliser la mémoire partagée pour diminuer les accès à la mémoire global plus coûteux,
4. utiliser la constant memory ou la texture memory pour avoir des lectures mémoire plus performantes,
5. recouvrir les accès mémoire avec du calcul.

Ces méthodes ont eu des résultats différents. La première a eu de bons résultats mais il n'y avait quasiment pas de répétition parmi les accès mémoire. Ce qui signifie que je n'ai pas eu des gains énormes et que ces gains sont restés localisés à une ou deux routines. La deuxième méthode était déjà appliquée pour la plupart des accès. Les accès qui ne sont pas contiguës en mémoire ne peuvent pas être modifiés sans rendre d'autres accès contiguës non contiguës ou en modifiant les algorithmes. Pour la mémoire partagée, les threads des routines utilisent chacun des données qui leur sont propres, la mémoire partagée est inutile pour ces routines. Sauf pour *fxroe5*, mais les accès mémoire sont trop chaotiques, l'utilisation de la mémoire partagée m'aurait pris trop de temps à mettre en place. La quatrième méthode n'est pas applicable dans notre cas. La texture memory ne fonctionne qu'en simple précision et la constant memory ne peut pas être allouée dynamiquement. Enfin pour la dernière méthode, le recouvrement, est fait de manière correcte par le compilateur sur ces routines. En effet les routines étudiées ne possèdent pas de boucles ou autres structures pouvant perturber le compilateur pour cette optimisation. De plus certaines routines (celles ayant les facteurs d'accélération les plus faibles) possèdent un grand nombre d'accès mémoire par rapport au calcul. Il n'y a pas assez de calculs pour permettre un recouvrement efficace.

Occupancy

L'occupancy est une notion représentant le taux d'utilisation du GPU (voir partie 3.1.2). Pour rappel, l'occupancy dépend du nombre de thread par bloc, de la taille de la mémoire partagée utilisée par bloc et du nombre de registres par threads. CUDA va essayer d'exécuter un maximum de blocs par multiprocesseur en même temps. C'est à dire 8 s'il y a assez de mémoire. Si les blocs consomment trop de mémoire (mémoire partagée ou registres) alors il devra limiter le nombre de blocs pour que leurs consommations mémoire tiennent dans la mémoire du multiprocesseur.

N'utilisant pas de mémoire partagée dans mes routines, il a fallu diminuer le nombre de registres utilisés et modifier la taille des blocs pour améliorer l'occupancy. Il y a un fichier excel sur le forum de NVidia permettant de trouver la meilleure taille de bloc à utiliser en fonction de la quantité de mémoire utilisée par les threads. En général le gain n'est pas très important mais sur 2 routines j'ai eu une diminution du temps d'exécution de 7/8%. Ce qui n'est pas si mal pour un faible coût. Ces routines avaient une occupancy inférieure à 18% avant l'optimisation, or ce pourcentage est important. C'est à partir de cette valeur que les dépendances de registres peuvent être recouverts par le calcul d'autres threads. Ce qui explique l'obtention d'un gain sur ces 2 fonctions.

Utilisation de la géométrie

Toutes mes routines étaient exécutées sur des blocs linéaires de N threads. Le nombre de threads par bloc fut modifié pour certaines routines lors de l'optimisation précédente mais les blocs restaient linéaires. Certaines routines possèdent des boucles (sur le nombre de dimensions, le nombre de flux...), j'ai supprimé ces boucles et ajouté une dimension au bloc pour remplacer les boucles. Seulement les routines concernées sont des routines n'ayant pas beaucoup de calculs et il s'est avéré que dérouler la boucle manuellement dans un thread, plutôt que de la découper en plusieurs threads, était plus efficace. Une seule routine a pu bénéficier de cette optimisation. Cette routine avait la particularité d'avoir certains accès mémoire entre deux itérations. Le déroulage a permis au compilateur d'optimiser ces accès mémoires. Cette optimisation aurait pu être effectuée dans le cadre des optimisations des accès mémoire, mais je n'y avais pas pensé en ce sens au début.

Le gain le plus important que j'ai obtenu est en supprimant des tests dans une routine. La routine faisait 6 tests d'affilé pour vérifier le nombre de flux traités puis modifiait le tableau des flux pour chaque flux (un peu comme une boucle for à incrémentations variables). J'ai supprimé tous les tests et à la place, j'ai mis le nombre de flux sur l'axe y de la géométrie des blocs. Ainsi chaque thread s'occupe d'une valeur pour un flux, plus besoin de vérifier le nombre de flux.

Pin memory

Lors de la deuxième stratégie de portage, certaines routines furent portées sur GPU. Deux d'entre elles furent remises sur CPU pour économiser le temps des transferts. Seul le résultat fut transféré. Cela a permis d'obtenir un gain de temps, mais un gain très faible. Pour essayer d'augmenter ce gain et tester l'efficacité de la Pin memory, qui devrait être plus rapide à transférer, j'ai externalisé ces deux transferts du système de gestion de la mémoire. Le calcul sur CPU fut modifié pour que le résultat soit écrit directement dans la Pin memory. J'ai ainsi testé la pin memory ainsi que le ZeroCopy (voir 4.1.1). Au final, le gain ne fut pas vraiment au rendez-vous. Le temps de transfert fut quasiment similaire entre la Pin memory et les transferts précédents. Pour le ZeroCopy, il semblerait que cela soit un peu plus rapide. Seulement il est difficile de mesurer le gain car, avec le ZeroCopy, le transfert est caché dans le temps d'exécution de la routine et au niveau de l'exécution total d'elsA, l'impact est très faible.

6.1.4 Portage en CUDA FORTRAN

Le portage, jusqu'à maintenant a été fait en CUDA C pour obtenir les meilleures performances. Mais les personnes utilisant et développant elsA sont surtout des chercheurs plus habitués à coder en Fortran qu'en C. Des tests entre le CUDA C et le CUDA Fortran avaient été effectués au début du stage et de nombreuses comparaisons étaient disponibles sur internet. Ces comparaisons relevaient un écart de performances entre les deux langages mais un écart assez faible. Cependant les performances sont très sensibles au code lui-même. Pour avoir une idée directement de la différence de performance sur des noyaux de calcul d'elsA, trois routines furent portées sur GPU.

La première difficulté fut d'intégrer l'utilisation du CUDA Fortran au moment de la compilation. Le CUDA Fortran étant un produit PGI et la compilation se faisant grâce à pgf90, on est passé des compilateurs INTEL aux compilateurs PGI. Ensuite, la gestion des bibliothèques dynamiques n'était pas encore prise en compte mais une version permettant la compilation dynamique fut disponible 2 semaines avant la fin du stage. Ce qui m'a permis de tester aussi la compilation dynamique mais au début la compilation devait être faite statiquement. Enfin des erreurs subsistaient lors de l'édition de lien. Les bibliothèques CUDA Fortran manquaient et le noyau de calcul CUDA FORTRAN subissait un renommage qui perturbait l'édition de lien. La documentation PGI sur CUDA Fortran n'offrait quasiment pas de renseignements sur la compilation. Au final, le problème de compilation fut résolu grâce à l'aide de leur hotline.

Par la suite j'ai porté 3 routines en CUDA FORTRAN. La première était une routine assez courte et simple que j'ai utilisée pour vérifier le bon fonctionnement de la compilation. Ensuite, comme le but était de comparer l'efficacité du CUDA Fortran par rapport au CUDA C, j'ai porté deux fonctions déjà portées en CUDA C. Pour avoir une bonne idée des différences de performance en fonc-

tion du genre de fonction, la première routine portée fut *fxcRoe5* qui obtenait un bon facteur d'accélération et la deuxième fut *compGradSca* qui obtenait un facteur d'accélération faible.

La comparaison fut effectuée à l'aide de CUDAPROF et des fonctions de timing qui m'avaient permis de calculer les accélérations. Il n'y a aucune différence entre la version CUDA C et la version CUDA Fortran de *compGradSca*. Il y a, par contre, une différence sur la fonction *fxcRoe5*. Le CUDA C est un peu plus rapide que le CUDA Fortran. On passe d'un facteur d'accélération de 11/12 avec le CUDA C à 10/11 avec le CUDA Fortran. Soit une différence confirmée par CUDAPROF de 7%. Mais en passant du compilateur INTEL au compilateur PGI. C'est pourquoi il faut vérifier la différence sur l'ensemble de l'exécution d'elsA. On peut voir cette différence sur la courbe 6.4, on peut voir que les deux courbes sont très proches l'une de l'autre.

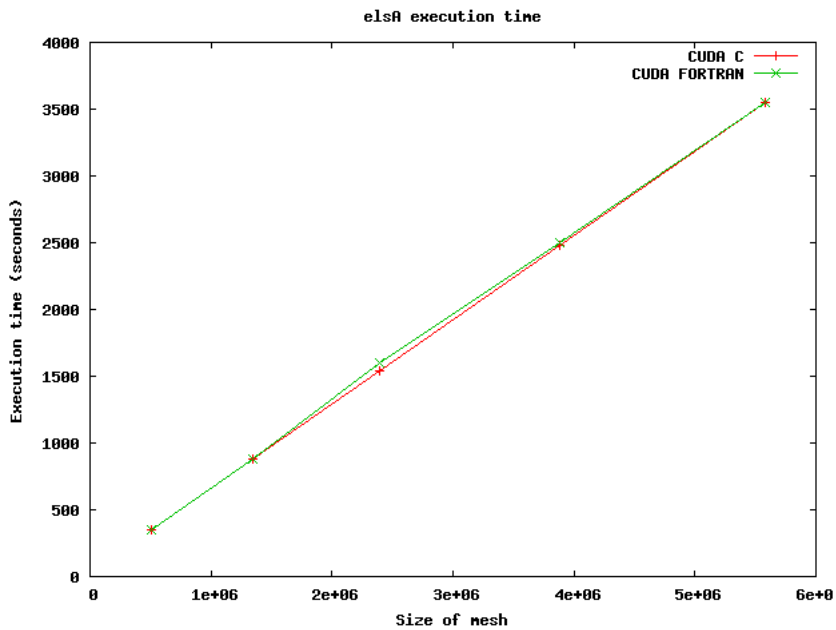


FIG. 6.4 – Mesures d'elsA CUDA C vs CUDA FORTRAN

6.2 Multi-Blocs et MPI

Jusqu'à maintenant, j'exécutais elsA sur un seul bloc et un seul processeur. Habituellement elsA effectue des calculs sur plusieurs blocs et utilise plusieurs

noeuds de calcul grâce à la bibliothèque MPI. Une fois les résultats sur le calcul monobloc analysés et qu'il était clair que pour avoir de bonnes performances il faudrait porter la quasi-totalité des routines sur GPU, l'étude sur le multi-blocs a pu commencer. J'ai commencé par faire fonctionner le multi-blocs avec la version GPU. Une fois la version multi-blocs fonctionnelle, je me suis penché sur le multi GPU avec l'utilisation de MPI.

6.2.1 Multi-Blocs

La première étape fut d'exécuter la version GPU d'elsA avec plusieurs blocs. Les routines portées sur GPU forment, lors de l'exécution, une séquence de calculs itérée 4 fois dans l'itération principale (celle que l'on fait varier). Il n'y a pas de changement de bloc à l'intérieur de cette séquence et à la fin de la séquence toutes les données sont rapatriées sur CPU, sauf les données temporaires qui ont déjà été libérées et les constantes. Malgré cela, les résultats obtenus étaient faux. En effet, les données constantes qui sont envoyées une seule fois sur GPU correspondent à un seul bloc. Il faut donc les modifier lorsque l'on change de bloc.

Par la suite, une libération de toute la mémoire GPU utilisée fut ajoutée à la fin de la séquence. Ainsi à chaque fois que l'on change de bloc, le GPU est vierge de données et les données du nouveau bloc sont envoyées sur le GPU. De cette manière le calcul multi-blocs, avec la version GPU d'elsA, était fonctionnel et correct. Le problème de cette solution est que les données constantes doivent être retransférées à chaque changement de bloc. Le temps perdu dans ces transferts superflus est élevé. La solution est loin d'être satisfaisante.

La troisième étape a consisté à rendre la version GPU multi-blocs satisfaisante. Pour se faire j'ai modifié mon système de gestion de la mémoire sur GPU. Jusqu'à maintenant j'utilisais la même mémoire GPU pour chaque bloc. C'est pourquoi il fallait la réinitialiser. Pour résoudre ce problème de manière correct, sans ajouter de transfert, j'ai ajouté les classes **GPU_Bloc_Vars** et **GPU_Bloc** au système de gestion de mémoire GPU. Ces classes permettent d'encapsuler le système de gestion de la mémoire précédent, qui possédait une instance de **GPU_Var** pour chaque donnée sur GPU, et d'avoir une encapsulation par bloc. Ainsi chaque bloc a sa propre liste de **GPU_Var** c'est à dire que les blocs ne partagent plus la même mémoire sur GPU. Pour plus de détail voir la section 6.3.

6.2.2 Multi-GPU

Une fois la version GPU multi-blocs fonctionnelle et correcte au niveau du résultat et du temps d'exécution, le développement de la version multi GPU a commencé. De la même manière que pour la version multi-bloc, une première exécution d'elsA GPU fut effectuée avec MPI. La encore, le résultat obtenu était faux. Après avoir trouvé où se trouvaient les communications MPI, j'ai activé les

messages de debug de la classe gérant ces communications. Malgré cela je n'ai trouvé aucune communication ou préparation de communication (empilement des zones mémoire à échanger) dans la séquence de calculs contenant les routines GPU. J'ai vérifié avec totalview mais la encore je n'ai trouvé aucune action liée aux communications dans la séquence de calculs GPU.

Il y avait pourtant bien un problème. Pour cibler le problème, j'ai désactivé toutes les routines GPU. Puis je les ai réactivées une à une en vérifiant à chaque fois le résultat de l'exécution. Au final, il s'est avéré que le rapatriement d'une donnée sur GPU était nécessaire au bon fonctionnement. Cette donnée étant le tableau `fxd`, voir deuxième stratégie de portage 6.1.2, qui est une variable temporaire sur GPU. Seul un rapatriement est nécessaire, pas besoin de la retransférer vers le GPU. Pour être sur que le problème venait bien d'une communication, j'ai déplacé le rapatriement du tableau `fxd` jusqu'à obtenir à nouveau une erreur de calcul. Au final, il faut rapatrier la donnée juste avant une routine Fortran pour avoir un résultat correct. Si le rapatriement est effectué après, le résultat devient incorrect. Ce qui est des plus étrange puisque la routine Fortran n'utilise pas `fxd` et n'effectue aucune action en relation avec MPI.

La routine fortran était la deuxième routine calculant directement le résultat dans de la Pin memory (voir la partie 6.1.3). Or, utilisant le système de Zero-Copy, cette mémoire pin est partagée entre le CPU et le GPU. Et ces deux routines Fortran sont suivies par une routine GPU recopiant les données de cette Pin memory dans le tableau `fxd` de la mémoire globale GPU. Effectuer le transfert de `fxd` juste avant cette routine forçait l'application à attendre la fin de l'exécution de la routine GPU précédente. Il y avait donc un problème d'accès concurrent à cette Pin memory. Au final, le transfert est inutile, il suffit d'ajouter une fonction de synchronisation juste avant de modifier la Pin memory.

6.3 Gestion de la mémoire

Lors du portage, il s'est vite avéré que l'on perdait beaucoup de temps dans les transferts de données entre CPU et GPU. Le but étant de minimiser les transferts de données et surtout d'annuler les transferts inutiles. Pour cela j'ai créé un système de gestion de la mémoire GPU dans la partie C++ du code d'elsA.

6.3.1 Les classes

Plusieurs classes furent créées pour gérer la mémoire sur GPU. Les principales classes sont présentes sur le diagramme 6.5. La première classe créée fut la classe **GPU_Var**. Cette classe permet de gérer la mémoire GPU pour une donnée. Il faut commencer par l'initialiser en configurant la taille de la donnée et le pointeur CPU. Puis on utilise les méthodes de la classe pour allouer, transférer et libérer la mémoire. Des flags sont utilisés pour donner le statut de la mémoire

GPU et les actions précédentes sont effectuées que si nécessaire.

Ensuite, durant le développement de la version multi-bloc, les deux classes **GPU_Bloc_Vars** et **GPU_Bloc** furent ajoutées au système. Comme on l'a vu dans la section 6.2, il a fallu ajouter un moyen pour que chaque bloc ait sa propre mémoire sur GPU. Ainsi la classe **GPU_Bloc_Vars** encapsule tous les objets **GPU_Var** nécessaires (un par donnée utilisée sur GPU). Elle représente les données d'un bloc. La classe **GPU_Bloc** permet de gérer une liste de **GPU_Bloc_Vars**. On aura ainsi un objet **GPU_Bloc_Vars** par bloc. Au final on a le fonctionnement représenté par le schéma 6.6. La gestion de la mémoire est totalement séparée des différentes fonctions C/C++ faisant appel aux routines GPU.

Une autre classe, qui n'est pas présente sur le diagramme 6.5, fut créée. Cette classe, **GPU_Context_Manager** permet de gérer les échanges du contexte GPU (voir section 4.1.2) entre threads. Le but était de permettre de recouvrir les transferts avec du calcul. Lors de la formation HMPP, les transferts asynchrones ont été abordés. En effet, certaines directives HMPP permettent d'effectuer des transferts asynchrones. Par contre, les développeurs de CAPS n'ont pas voulu utiliser la pin memory qu'ils jugent trop contraignante. Pour simuler de l'asynchronisme dans leurs transferts, ils utilisent des threads. J'ai essayé de reproduire cela dans elsA. Cette classe contient des méthodes et attributs statiques permettant de libérer et de prendre le contexte. L'idée étant que lors d'un appel à un transfert asynchrone, le processus libère le contexte. Le thread créé récupère le contexte pour avoir accès à la même mémoire GPU et effectue le transfert. Si un autre transfert asynchrone est effectué, alors le thread créé se met en attente du contexte. Enfin le processus principal récupère le contexte quand tous les transferts asynchrones sont terminés. Cependant j'obtiens une erreur lors du passage de contexte que je n'ai pas réussi à résoudre. Le contexte passe bien du processus au thread mais lorsque le thread libère le contexte, pour que le processus ou un autre thread puisse le récupérer, il semblerait que le contexte soit détruit. Cela provoquerait l'erreur lorsque le processus principal se met en attente du contexte.

6.3.2 Les transferts effectués

Malgré le système de gestion, le nombre de transferts reste assez élevé. De plus le système de gestion aide surtout à savoir si la mémoire est déjà allouée ou non. En effet, elle ne permet pas de savoir si la mémoire a été modifiée du côté CPU ou du côté GPU. La plupart des optimisations des transferts sont effectuées manuellement. Les transferts mémoire sont représentés sur le schéma 6.7. Les lignes de couleurs signifient que la donnée est allouée sur le GPU et les flèches représentent les transferts. On retrouve la séquence de routines effectuées 4 fois par itération principale. On peut voir qu'il reste encore beaucoup de transferts et que, à part les constantes (surf, vol et snorm), aucune donnée n'est réutilisée directement d'une séquence à une autre. Il faut retransférer les données

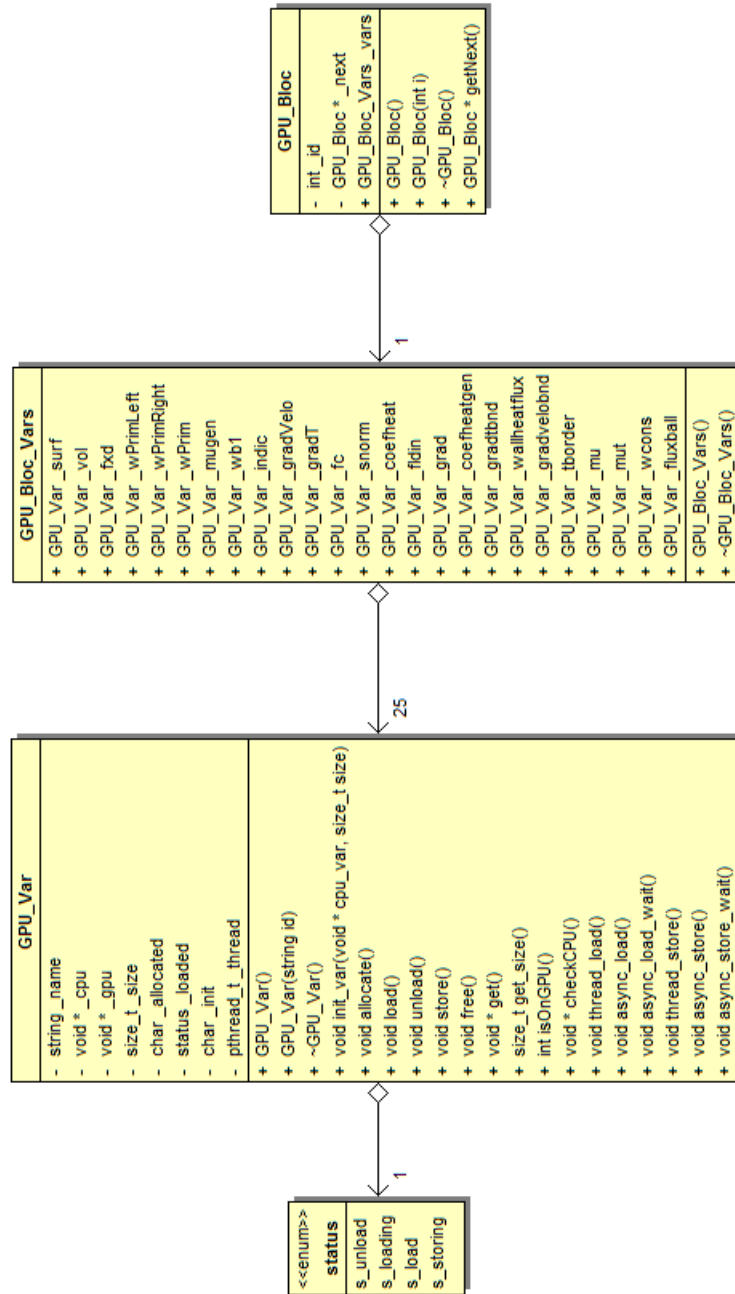


FIG. 6.5 – Diagramme des classes de gestion de la mémoire GPU

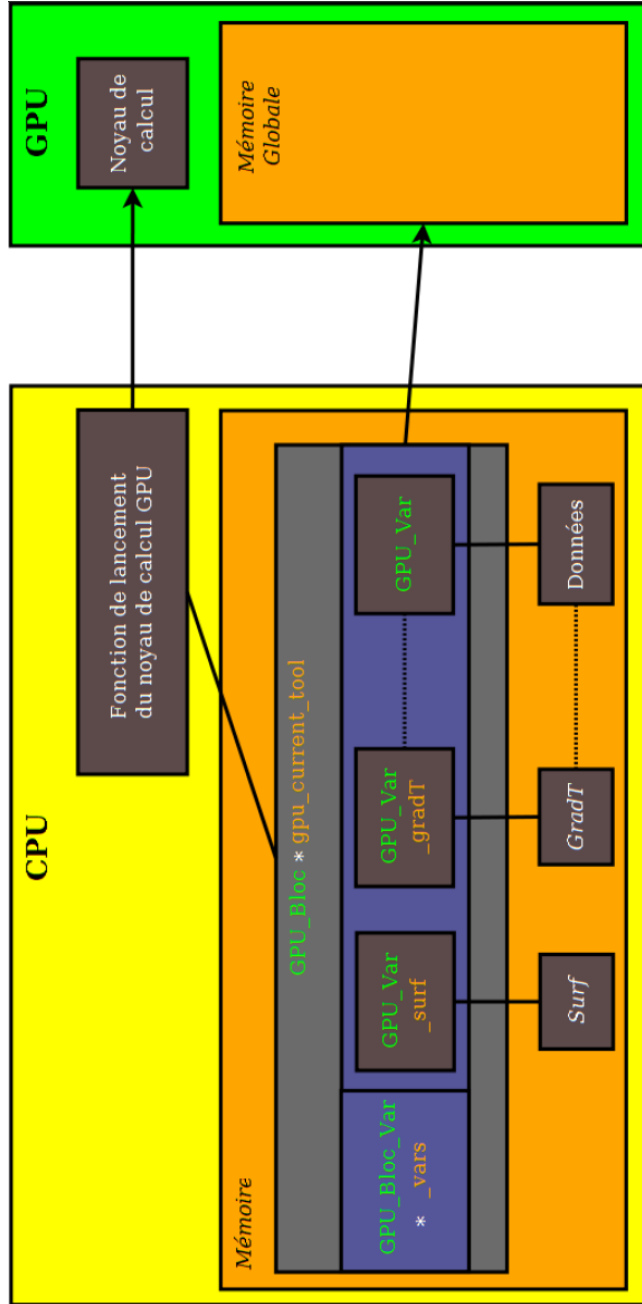


FIG. 6.6 – Gestion de la mémoire GPU

nécessaires à chaque début de séquence et les rapatrier à la fin.

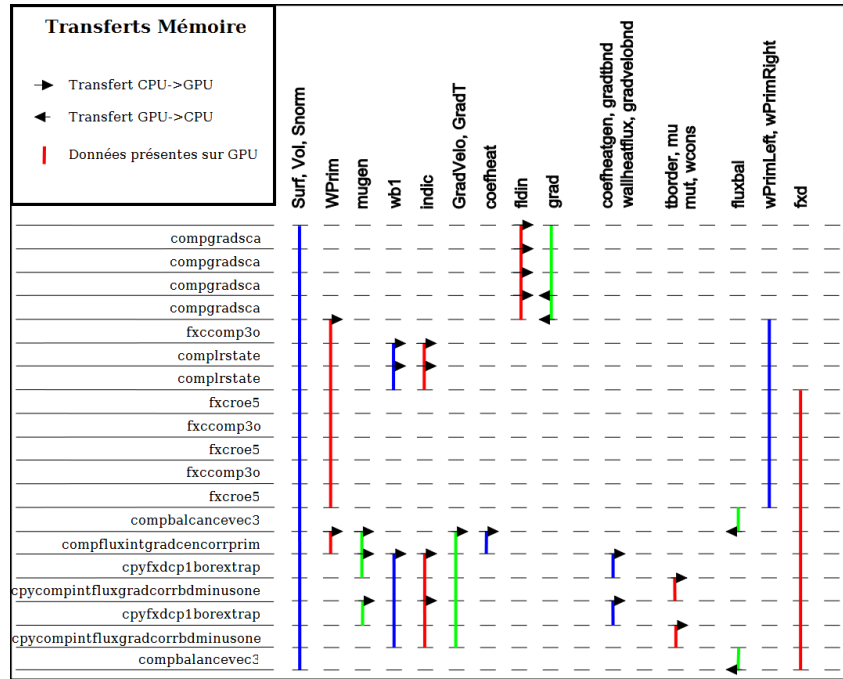


FIG. 6.7 – Transferts mémoire

Chapitre 7

Résultats

7.1 Matériel pour les mesures

Le CERFACS a reçu deux GPU pendant mon stage que j'étais seul à utiliser. La première a été reçue un mois après le début du stage, c'est une tesla C1060 possédant 30 multiprocesseurs de 8 coeurs (240 coeurs de 1.30Ghz) et 4Go de mémoire globale. Le deuxième est un carte FERMI et a été reçue fin mai. C'est une tesla C2050 possédant 14 multiprocesseurs de 32 coeurs (448 coeurs de 1.15Ghz) et 3Go de mémoire globale. Les deux cartes sont sur deux machines semblables nommées tesla et tesla2 ayant un Intel Xeon "Nehalem" 2.00 Ghz comme CPU. Pour les mesures CPU, j'ai eu accès au serveur de calcul Octopus, serveur utilisé par la plupart des utilisateurs d'elsA. Ce serveur est composé de CPU Intel Xeon "Nehalem" 2.66Ghz.

Sinon j'ai également eu accès à deux serveurs de calcul extérieurs possédant des GPU dès le début du stage. Le premier, au CINES, est composé de tesla S1070. La tesla S1070 est une carte spécifique au serveur de calcul. Elle contient 4 GPU équivalents à la C1060 mais avec seulement 4Go de mémoire globale par GPU. Chacune des S1070 est connectée à deux noeuds. Ce qui signifie qu'il y a deux GPU par noeud et que les deux GPU doivent partager le débit du câble PCI. Chaque noeud possède aussi 2 quadri-coeurs Intel Xeon 5472 cadencés à 3.00 Ghz. Le deuxième serveur est un serveur du CEA. Les noeuds sont composés carte FX5800 (1,29 GHZ, 4Go mémoire globale) et de 2 processeurs Intel Xeon 5450 quadri-coeurs cadencés à 3 Ghz.

7.2 Performances

7.2.1 Par routines

La première information importante est de connaître le gain d'accélération obtenue sur chaque routine de calcul. Cela permet de connaître les véritables capacités d'accélération des GPU sans transferts. Les accélérations présentées dans le tableau 7.1 ont été mesurées avec la version GPU sur la tesla C1060 et C2050 du CERFACS. Le temps témoin étant la temps d'exécution sur 1 coeur d'Octopus. Comme on peut le voir, les accélérations sont assez hétérogènes. On passe d'un gain de 15/20 pour *TurCompDiffFluxDensIntCorr* à 3/4 pour *complrstate*. Le gain dépend énormément de la routine. Pour des routines ayant beaucoup de calculs à faire et peu d'accès mémoire par rapport au nombre de calculs on aura des accélérations proches de celle de *TurCompDiffFluxDensIntCorr*. Pour les routines ayant très peu de calculs et beaucoup d'accès mémoire, les accélérations seront plus proches de celle de *complrstate*. En effet, s'il n'y a pas assez de calculs, le GPU ne pourra pas recouvrir les accès mémoire avec du calcul et on se retrouvera avec des routines qui passeront la plus grande partie de leur exécution à attendre la mémoire.

Ensuite si on étudie le passage de la C1060 à la C2050, on peut remarquer qu'une fonction a été ralentie. *complrstate* est passée d'un facteur d'accélération de 4/5 à 3/4. Pourtant l'architecture FERMI est censée améliorer les performances. Cependant elle ajoute de nombreuses fonctionnalités qui ont un coût en temps. La plus importante étant l'ECC (Error Correcting Code). Il y a ensuite d'autre fonctionnalités qui peuvent ralentir l'exécution comme le passage à des pointeurs 64bits. Donc il suffit qu'une routine ne bénéficie pas ou très peu des améliorations de la FERMI pour qu'elle soit ralentie. Ce qui est le cas dans *complrstate*. C'est une petite routine qui contient un bon nombre d'accès mémoire mais chaque accès est effectué sur un espace mémoire différent et ne bénéficie pas du système de cache (ou pas suffisamment). D'ailleurs des mesures ont été effectuées avec l'ECC désactivé et toutes les routines sont accélérées. Plus particulièrement *complrstate* qui retrouve un temps d'exécution comparable à celui obtenue sur la C1060.

Ensuite toutes les accélérations ont un peu augmenté. Sachant que les routines ayant pour facteur limitant les accès mémoire ne bénéficient pas pleinement de l'augmentation du nombre de coeurs double précision. Enfin la routine *fxcroe5* a subi une bonne accélération entre les deux GPU (x2). Cette routine a un certain nombre d'accès mémoire communs entre les threads. La mémoire partagée aurait pu être utilisée pour cette routine mais les accès étaient assez difficiles à regrouper par bloc. Problème contourné par l'architecture FERMI grâce au système de cache. Ce qui explique cette accélération importante par rapport aux autres.

Facteurs d'accélération des routines					
Mesh size		511485	1349985	2396485	3884985
TurCompDiffFluxDensIntCorr	C1060	14.32	12.39	12.61	14.32
	C2050(FERMI)	19.71	17.15	17.79	19.64
fxcroe5	C1060	8.87	8.99	8.74	8.71
	C2050(FERMI)	19.92	20.32	19.86	19.82
compbalancevec3	C1060	7.76	7.32	7.32	7.84
	C2050(FERMI)	11.31	10.41	10.44	11.13
fxccomp3o	C1060	8.3	5.66	5.48	6.09
	C2050(FERMI)	8.58	5.6	5.57	6.08
compgradsca	C1060	3.83	3.93	3.91	3.94
	C2050(FERMI)	7.72	6.62	6.54	6.74
complrstate	C1060	4.89	5.12	5.37	5.77
	C2050(FERMI)	3.45	4.25	4.62	4.78

TAB. 7.1 – mesures d'accélération des routines

7.2.2 Sur la totalité d'elsA

Maintenant que l'on connaît l'accélération par routine, étudions les résultats sur l'exécution totale de l'application sur le cas test. On va commencer par étudier le profiling de la version GPU d'elsA. Sur la figure 7.1, on a le profiling d'elsA GPU fourni par CudaProf (outils NVidia). Ce profiling ne prend en compte que les routines exécutées sur le GPU. On peut voir sur le profiling, comme on l'a déjà vu pendant le portage, que les transferts mémoire prennent encore plus de 55% du temps d'exécution sur GPU. Ce profiling a été effectué sur la C1060, le problème est encore pire sur la C2050 où les routines ont été accélérées.

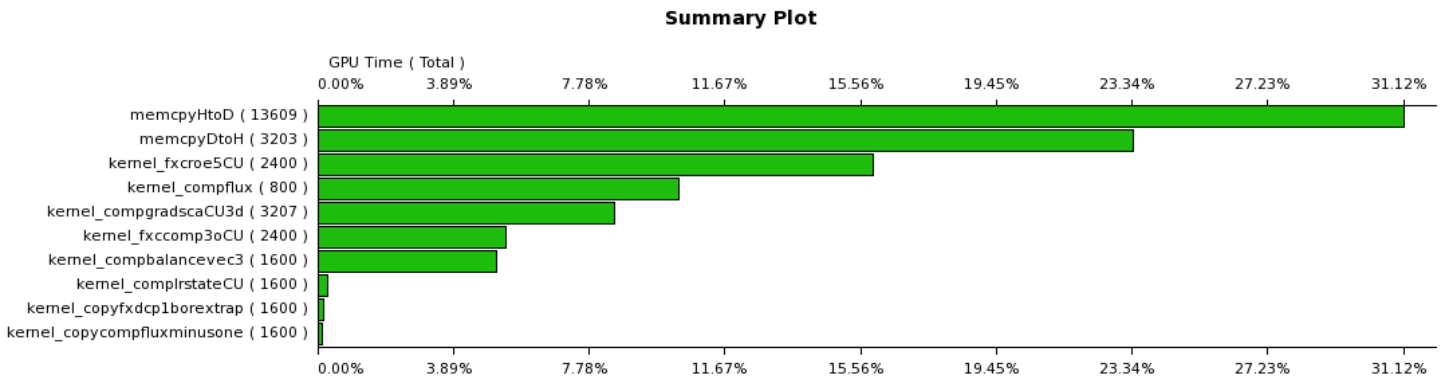


FIG. 7.1 – Profiling CudaProf (Taille : 65x329x361, Iter : 200)

Le graphique 7.2 représente le temps d'exécution d'elsA sur le cas test en pourcentage par rapport au temps d'exécution de la version CPU d'elsA sur la machine tesla (CPU Intel Xeon 2.00Ghz). Il y a, sur le graphique, 3 courbes différentes : l'exécution de la version CPU sur Octopus et de la version GPU sur la carte C1060 et C2050. Déjà, on voit que l'exécution sur Octopus est plus rapide que sur la machine tesla puisque l'on passe d'un CPU à 2.60Ghz à un CPU à 2.00Ghz. Ensuite sur l'exécution des versions GPU, on a une accélération d'un facteur légèrement supérieur à 2. Par contre, il y a très peu de différences entre les deux cartes puisque les transferts de mémoire ne sont pas améliorés entre les deux. Maintenant si on compare avec Octopus, on n'a plus notre facteur 2 mais on s'en rapproche malgré tout.

De plus, il ne faut pas oublier que les routines portées sur GPU représentent 67% du temps d'exécution d'elsA sur CPU. Cela signifie que 33% du temps d'exécution d'elsA ne sera pas accéléré. La barre noire représente la séparation entre les routines portées sur GPU et les autres routines. Sur les versions GPU cette barre est à 33% puisque la partie CPU est exécutée sur la même machine que la mesure de référence. Ainsi les valeurs pour la C1060 et C2050 ne pourront jamais descendre en dessous de cette barre noire. Pour les mesures Octopus, cette barre est en dessous, puisque ces autres routines sont aussi accélérées par rapport à la mesure de référence. Il faut prendre cela en compte dans l'optique d'un serveur de calcul mixte GPU/CPU avec des GPU et des CPU performants. Sur un tel serveur on retrouvera notre facteur 2 par rapport à Octopus.

7.2.3 Potentiel

Un stage de 7 mois est trop court pour pouvoir porter toutes les routines sur GPU. Cependant, c'est ce qu'il faudrait faire pour commencer à avoir des accélérations conséquentes. Malgré tout, même si le stage est trop court pour obtenir cette accélération, on a obtenu assez d'éléments pour conjecturer une valeur d'accélération atteignable. Grâce aux valeurs déjà obtenues, on va calculer le gain que l'on peut espérer atteindre si on porte toutes les routines sur GPU. Sachant que cela annulera les transferts mémoire (sauf au début et à la fin de l'exécution) et accélérera les dernières routines qui sont encore sur CPU. Pour le calcul de ce gain potentiel, j'ai utilisé les profilings du maillage de taille 65x229x261 sur 200 itération. Ces profilings sont fournis en annexes. Pour le calcul, j'ai suivi les équations suivantes : Temps des routines sur CPU :

$$T_1 + T_2 + T_3 + T_4 + T_5 + T_6 = T \quad (7.1)$$

Temps des routines sur GPU :

$$T'_1 + T'_2 + T'_3 + T'_4 + T'_5 + T'_6 = T' \quad (7.2)$$

Accélération par routines :

$$\forall i \in [1, 6], a_i = \frac{T_i}{T'_i} \quad (7.3)$$

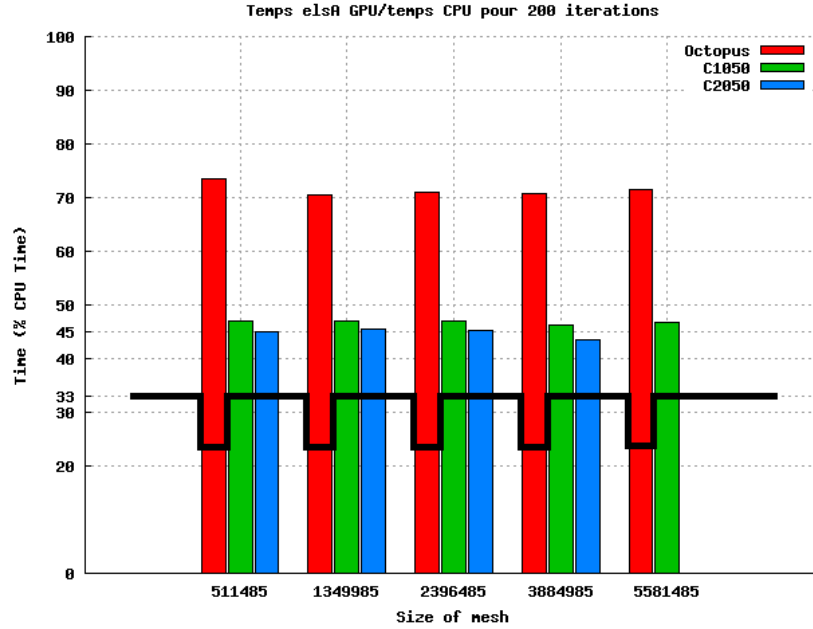


FIG. 7.2 – Accélération d'elsA

Accélération globale :

$$A = \frac{T}{T'} \quad (7.4)$$

$$A = \frac{T'_1 + T'_2 + T'_3 + T'_4 + T'_5 + T'_6}{T} = \sum_{i=1}^6 \left(\frac{1}{a_i} \frac{T_i}{T} \right) = \sum_{i=1}^6 \left(\frac{1}{a_i} x_i \right) \quad (7.5)$$

Avec x_i =pourcentage de temps passé dans la routine i

Pour calculer le gain potentiel, j'ai repris ces formules. J'ai ajouté un T_7 représentant toutes les autres routines. Tous les T_i sont connus mais T'_7 et a_7 ne sont pas connus. Partant du principe que les bonnes routines ayant beaucoup de calculs ont déjà toutes été portées et que celles qui restent à porter seront plus proches des petites routines tel que *complrstate*, la valeur moyenne d'accélération pour les routines restantes fut mise à 3 pour la C1060 et à 4 pour la C2050. Une fois a_7 connu, on connaît T'_7 , T' et on peut calculer A, l'accélération totale que l'on recherche, avec l'équation 7.4. On peut aussi calculer A avec l'équation 7.5, cela permet de vérifier le résultat.

On peut retrouver les résultats du calcul sur le tableau 7.2. Le sous total correspond au gain que l'on a actuellement sur l'exécution des 6 routines uniquement et sans les transferts mémoire. Ensuite on retrouve les facteurs

	Accélération C1060	Accélération C2050	Pourcentage sur la totalité d'elsA	Pourcentage sur l'ensemble des 6 routines
turCompDiffFluxDensIntCorr	12	16.41	25.29%	36.78%
fxcroe5	7.47	17.94	24.66%	35.86%
compbalancevec3	6.39	9.01	6.94%	10.09%
fxcomp3o	4.97	4.97	5.81%	8.45%
compgradsca	3.19	5.34	5.80%	8.44%
complrstate	4.81	4.02	0.26%	0.38%
sous total	7.21	11.45	68.76%	100.00%
autres routines	3	4	31.24%	0.00%
total	5.01	7.24	100.00%	100.00%

TAB. 7.2 – Calcul du gain potentiel

d'accélération de 3 et de 4 choisis pour représenter l'accélération des routines non encore portées. On obtient ainsi une accélération potentielle de 5.01 sur la C1060 et de 7.24 sur la C2050.

7.3 Multi-blocs

7.3.1 Mono GPU

La question, lors du passage du monobloc au multi-blocs, était de savoir si le gain restait le même en ajoutant plusieurs blocs ou si la version GPU allait subir une détérioration trop grande des performances. Comme on l'a vu dans la section 6.2 le passage du monobloc au multi-blocs n'a pas vraiment ajouté de traitements supplémentaires. Bien sur, on traite plusieurs blocs avec ce que cela implique : multiplication du nombre d'appels aux routines de calculs et du nombre de transferts mémoire. Mais au final, chaque bloc effectue le même nombre de transferts que s'il était traité en monobloc et la quantité de données transférée reste la même. Il n'y a aucune raison pour que les performances soient plus détériorées que dans la version CPU.

Les résultats se trouvent sur les courbes 7.3 et 7.4. Sur la première courbe on a le temps d'exécution d'elsA en fonction du nombre de blocs. On peut voir que les courbes sont presque parallèles. La courbe d'Octopus a une pente plus faible que celle des autres. Le surcoût du multi-blocs est principalement causé par la partie CPU de l'exécution et le CPU est plus rapide sur Octopus. De plus, le fait de faire plusieurs petits transferts plutôt qu'un gros transfert, rajoute aussi un coût supplémentaire, même si ce dernier est assez faible. La deuxième courbe représente le rapport des temps d'exécution. Si on regarde les courbes de rapport entre les deux versions GPU et le CPU de tesla, la courbe augmente mais assez légèrement. Elle augmente plus sur les courbes dont le rapport est

effectué avec le temps Octopus. Mais la encore, si on oublie le dernier point, l'augmentation reste assez faible. Au final le multi-blocs reste correct.

Les courbes pour les autres tailles ne sont pas présentes car j'ai corrigé une erreur modifiant les temps d'exécution en fin de stage. Je n'ai malheureusement pas eu le temps de retracer les courbes pour les autres tailles. Cependant, Les autres tailles avaient des courbes similaires. Sauf la courbe pour la plus petite taille (511485 points) qui avait une différence plus importante dans les pentes. La courbe subissait une croissance plus importante en fin de courbe. Cela peut s'expliquer, par le fait que l'on découpe en sous blocs un bloc déjà petit. On se retrouve au final avec plusieurs petits blocs. Pour obtenir les meilleures performances sur GPU il faut suffisamment de calculs pour bénéficier au mieux du nombre important de coeurs de calcul. Lorsque les blocs commencent à devenir trop petits, les performances chutent. Nous n'avons pas vu ce phénomène jusqu'à maintenant, car les tailles de blocs utilisées couramment avec elsA sont suffisamment grandes pour ne pas rencontrer ce problème.

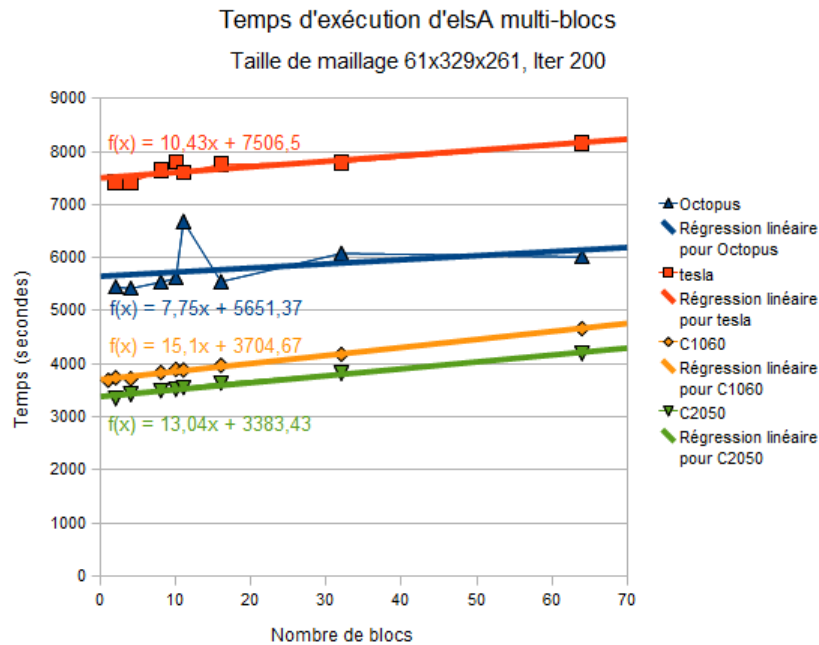


FIG. 7.3 – Temps d'elsA multi-blocs

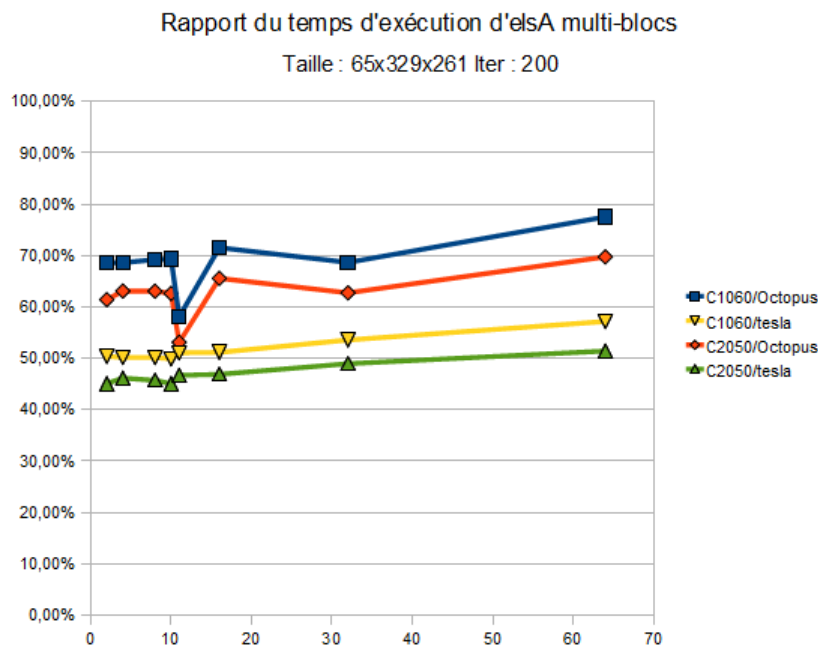


FIG. 7.4 – Rapport des temps d'elsA multi-blocs

7.3.2 Multi GPU

Comme pour le multi-blocs, que l'on vient de voir, le passage au multi GPU ne devrait pas ajouter de temps de traitement par rapport au passage à la version multi noeuds sur CPU. Les mesures GPU et CPU, de cette partie, ont été effectuées sur le serveur du CEA. Des mesures auraient du être effectuées aussi sur le serveur du CINES, mais une erreur de communication MPI apparaît sur certaines exécutions, et je n'ai pas eu le temps de regarder en détail cette erreur.

Par rapport au multi-blocs, les courbes pour le multi GPU sont beaucoup plus parallèles. Cela est visible sur la courbe 7.5. Les rapports de temps d'exécution sont aussi beaucoup plus constants. Ces rapports sont présents sur la courbe 7.6. On peut voir, sur cette courbe, le même phénomène que pour le multi-blocs, la courbe bleue subit une baisse de performances en passant de 8 blocs à 10 blocs.

La encore, le temps m'a manqué pour bien faire varier le nombre de GPU utilisés. Mais le cas test et les scripts utilisés ont été rassemblés dans un fichier organisé, avec un README expliquant comment utiliser les différents scripts et versions du cas test. Ainsi ces mesures pourront être menées facilement après

mon départ. De plus NVidia offrait un accès à un de leur serveur de calculs composé de cartes FERMI. A cause de problème administratif entre NVidia et l'ONERA, pour pouvoir mettre le code d'elsA sur le serveur de NVidia, je n'ai pas eu accès à ce serveur à temps. Mais quand le serveur sera disponible, les mesures pourront aussi être effectuées sur ce serveur.

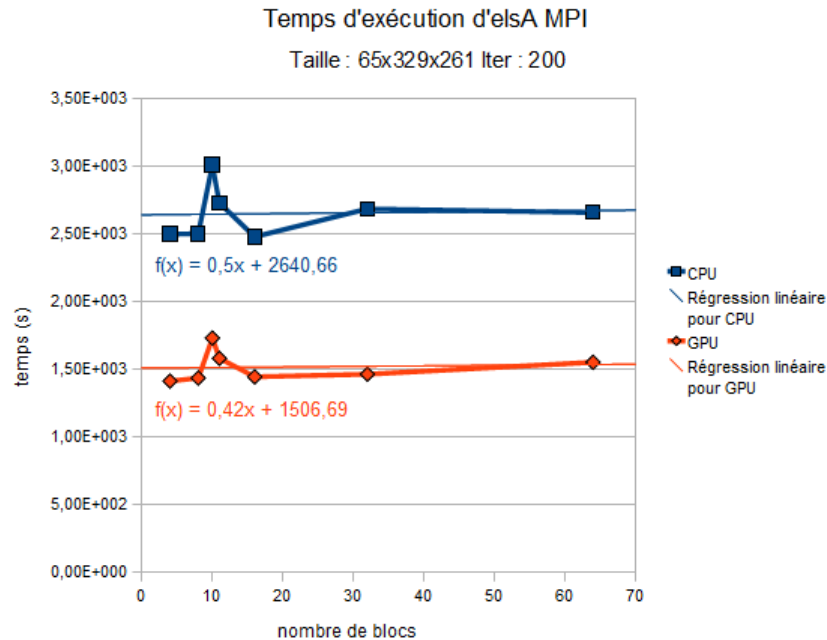


FIG. 7.5 – Temps d'elsA 4 GPU

7.4 Validation des résultats

Le calcul est plus rapide sur GPU que sur CPU, mais encore faut-il vérifier qu'il est correct. Pour valider les résultats, j'ai comparé le fichier résidu produit par elsA à la fin d'une exécution ainsi que les fichiers flow contenant un certain nombre de données du problème à la fin du calcul. J'ai commencé par faire la différence entre les deux fichiers mais la valeur obtenue n'est pas vraiment significative. Une différence de 1 peut être énorme pour certains types de données et négligeable pour d'autres. C'est pourquoi, par la suite, la différence en pourcentage fut calculée. Mais il y a un certain nombre de valeurs proches de zéro dont la valeur reste négligeable par rapport à l'ensemble des données. Cependant ces valeurs obtenaient des différences en pourcentage énorme ($\approx 1000\%$). Après conseil des utilisateurs et développeurs d'elsA, je suis revenu à la différence ab-

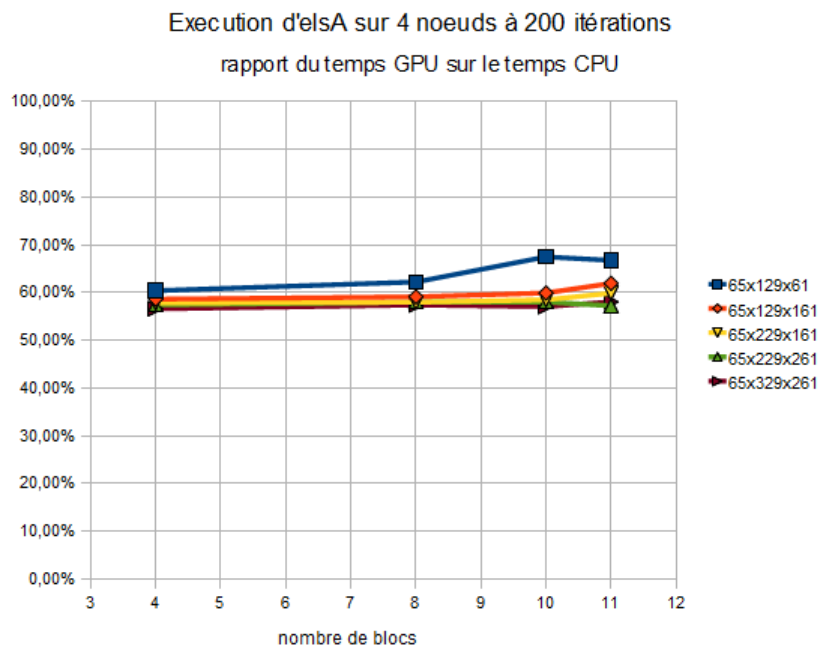


FIG. 7.6 – Rapport des temps d'elsA 4 GPU

solue. Le résultat des comparaisons des résidus est représenté par la courbe 7.4. Pour référence, il y a la courbe 7.4 qui représente la même comparaison des résidus entre la version CPU Intel et la version CPU PGI. Les erreurs sur GPU sont un petit peu plus importantes mais restent correctes. Il n'y a pas d'erreur conséquente qui justifierait de rejeter les résultats. La comparaison GPU a été effectuée sur la carte C2050, mais le taux d'erreur est quasiment le même sur C1060.

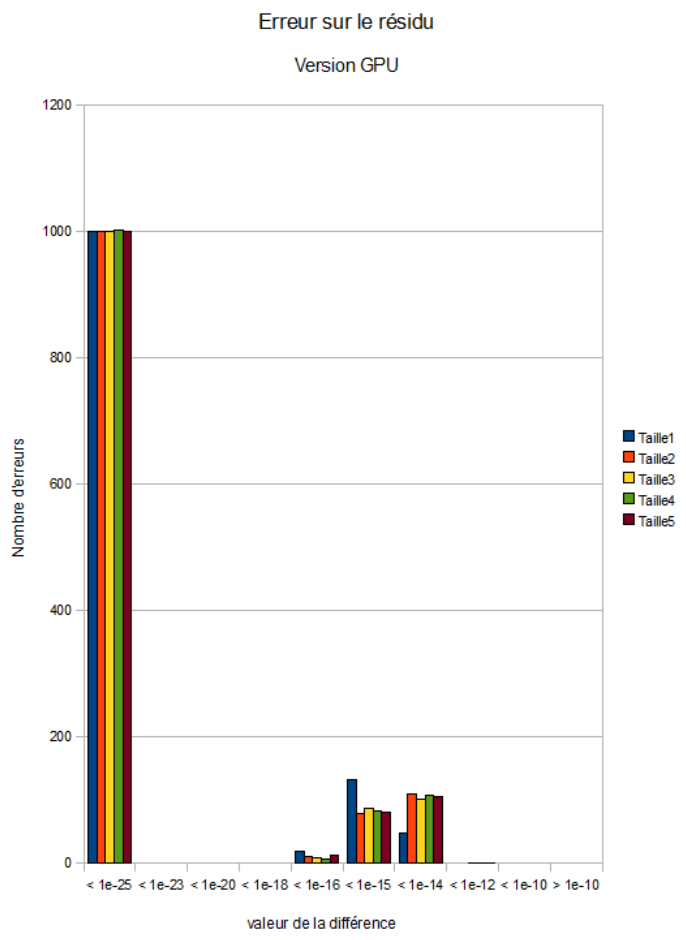


FIG. 7.7 – Taux d’erreurs du calcul GPU par rapport à la version CPU INTEL

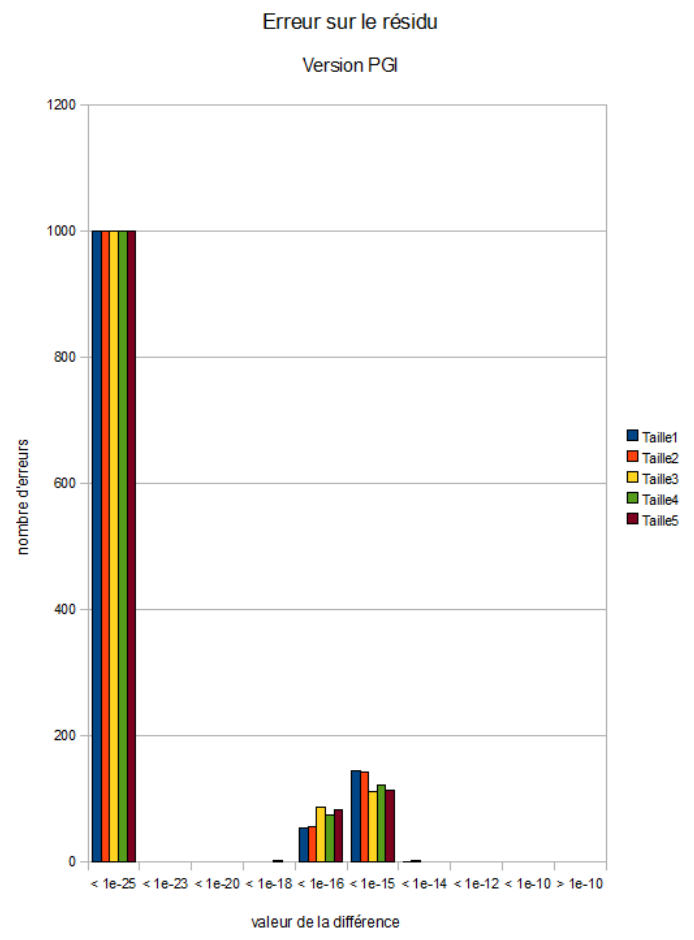


FIG. 7.8 – Taux d’erreurs du calcul CPU PGI par rapport à la version CPU INTEL

Chapitre 8

Conclusion

Durant le stage, j'ai beaucoup étudié les GPU. Ces derniers représentent une nouvelle solution pour le calcul haute performance. Cela permet d'avoir un petit ordinateur directement dans sa propre machine de bureau. De plus son haut niveau de parallélisme lui permet d'obtenir des pics de performances importants. Cependant cela rend aussi son utilisation plus difficile et spécifique. En effet, il faut pouvoir extraire un niveau de parallélisme très fin pour pouvoir utiliser toute la puissance d'un GPU. De plus, l'architecture particulière des GPU demande un certain temps d'adaptation aux développeurs. Mais l'utilisation des GPU dans le calcul haute performance est assez récente. La communauté autour de cette utilisation est déjà importante et très active. NVidia se montre à l'écoute de cette communauté et fait évoluer ses GPU pour les rendre plus simples d'utilisation et plus performantes, comme on peut le voir avec la nouvelle architecture FERMI.

Les résultats obtenus de l'utilisation des GPU pour accélérer l'application elsA sont plutôt prometteurs. Le facteur d'accélération 2 que l'on a obtenu est faible. Mais seulement 67% du temps d'exécution fut porté sur GPU et le temps perdu dans les transferts entre GPU et CPU est considérable. Si on règle ces deux problèmes en portant toutes les routines sur GPU alors le facteur d'accélération potentiel qui a été calculé devient tout de suite plus intéressant. Ceci, sans avoir à modifier l'algorithmique. Cependant le cas test avait été choisi pour ses accès mémoire contiguë convenant parfaitement au GPU. Il y a d'autres cas dans elsA où les accès mémoire sont plus dispersés ce qui nuira gravement aux performances GPU. C'est pourquoi il faut considérer ces résultats avec précaution. Le mieux serait de refaire la même chose avec un tel cas pour avoir une idée de la perte de performance occasionnée par ces accès mémoires dispersés. Surtout qu'avec l'architecture FERMI et son système de cache on peut espérer que cela limite la perte de performance. Enfin il y a beaucoup d'articles sur les accès mémoire dispersés expliquant comment limiter la perte de performance.

Ce stage m'a permis de découvrir un environnement entre la recherche et

l'industrie. J'ai beaucoup appris sur le monde de la recherche notamment en côtoyant les chercheurs et les étudiants en thèse. De plus j'ai pu confirmer mes connaissances obtenues en cours sur les problématiques de performance en découvrant que derrière cette problématique il y a aussi celle du coût. Et que dans ce coût il y a bien sur le coût du portage mais aussi du matériel et, de plus en plus, le coût de la puissance électrique nécessaire et du dégagement de chaleur. Enfin ce fut un stage très enrichissant durant lequel j'ai eu beaucoup de contacts en dehors du CERFACS : ONERA, Total, airbus, NVidia, CAPS, PGI.

Bibliographie

- [1] *NVIDIA CUDA C Programming Guide*.
- [2] *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture : FERMI*.
- [3] Florent Dahm. Etude pour l'accélération du code de calcul parallèle elsa à l'aide de processeurs graphiques (gpu). <http://www.ann.jussieu.fr/MathModel/rapports/FDahm.pdf>, 2008.
- [4] Maryam Sadooghi-Alvandi Henry Wong, Misel-Myrto Papadopoulou and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. <http://www.eecg.toronto.edu/~myrto/gpuarch-ispas2010.pdf>, Mars 2010.
- [5] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf, Novembre 2008.

Annexe A

Profiling

Profiling OCTOPUS Taille 4 Iter 200					
% time	cumulative seconds	self			name
		seconds	calls	s/call	
25.29	747.34	747.34	800	0	compfluxintgradcencorprim
24.66	1476.12	728.78	2400	0	fxcroe5
6.94	1681.31	205.19	1600	0	compbalancevec3
5.81	1853.02	171.71	2400	0	fxccomp3o
5.8	2024.3	171.28	3207	0	compgradsca2
4.24	2149.57	125.27	1603	0	operatordivf
4.09	2270.36	120.79	9604	0	cons2tempf
3.22	2365.41	95.05	2402	0	cons2speedf
2.85	2449.6	84.19	800	0	cons2prim
2.2	2514.46	64.86	2400	0	fldaddpart
2.02	2574.03	59.57	800	0	TmoSolverBase::computeLhs()
1.12	2607.25	33.22	800	0	lhsupdate
0.99	2636.37	29.12	2401	0	compmu
0.94	2664.06	27.69	6400	0	compgradborder
0.93	2691.66	27.6	1600	0	compintfluxgradcorbminusone
0.83	2716.22	24.56	800	0	cons2primts
0.79	2739.58	23.36	800	0	FldFieldF::mulsca_mul_assign
0.74	2761.43	21.85	201	0	cpdtconv
0.66	2780.96	19.53	6414	0	compbndgrad
0.64	2799.76	18.8	201	0	turcompmutwale
0.51	2814.85	15.09	6406	0	extrapof
0.47	2828.72	13.87	200	0	TmoSolverBase::solutionPEqualSolutionN
0.46	2842.23	13.51			spec_dexp2
0.36	2852.86	10.63	800	0	FldFieldF::operator!=(FldFieldF const&)
0.35	2863.11	10.25	2002	0	isothwallg
0.29	2871.8	8.69	1600	0	fxdcp1borextrap
0.29	2880.48	8.68	337885241	0	FldArrayF::operator[](int)
0.29	2888.95	8.47	201	0	cptimesteptur
0.26	2896.65	7.7	1600	0	complrstate
0.26	2904.33	7.68	4632852509	0	FldFieldF::operator[](int)
0.21	2910.57	6.24	800	0	FldFieldF::add_and_assign
0.17	2915.69	5.12	20024	0	joincopyfld
0.16	2920.44	4.75	800	0	FldFieldF::mulsca_add_assign
0.14	2924.52	4.08	802	0	FldFieldF::mulsca_and_assign
0.14	2928.51	3.99	800	0	SouForcTerm::compute
					2954.83

FIG. A.1 – Profiling de la version CPU sur Octopus

Profiling CPU Taille 4 Iter 200					
% time	cumulative seconds	self			name
		seconds	calls	s/call	
26.23	1241.17	1241.17	800	0	compfluxintgradcencorrprim
24.18	2385.1	1143.93	2400	0	fxcroe5
6.54	2694.72	309.62	3207	0	compgradsca2
6.51	3002.79	308.07	1600	0	compbalancevec3
6.01	3287.37	284.58	2400	0	fxccomp3o
3.9	3471.68	184.31	1603	0	operatordivf
3.82	3652.43	180.75	9604	0	cons2tempf
2.96	3792.41	139.98	2402	0	cons2speedf
2.62	3916.56	124.15	800	0	cons2prim
2.24	4022.78	106.22	2400	0	fldaddpart
1.87	4111.08	88.3	800	0	TmoSolverBase::computeLhs()
1.12	4164.3	53.22	800	0	lhsupdate
0.98	4210.54	46.24	6400	0	compgradborder
0.91	4253.43	42.89	1600	0	compintfluxgradcorrbdminusone
0.9	4296.22	42.79	2401	0	compmu
0.81	4334.32	38.1	800	0	cons2prints
0.78	4371.1	36.78	800	0	FldFieldF::mulsca_mul_assign
0.74	4405.88	34.78	201	0	turcompmutwale
0.68	4438.17	32.29	201	0	cpdtconv
0.67	4469.92	31.75	6414	0	compbndgrad
0.47	4492.28	22.36	6406	0	extrap0f
0.47	4514.49	22.21	800	0	SouForcTerm::compute
0.44	4535.1	20.61	200	0	TmoSolverBase::solutionPEqualSolutionN
0.42	4555.1	20			spec_dexp2
0.42	4574.93	19.83	4632852509	0	FldFieldF::operator[]
0.33	4590.67	15.74	2002	0	isothwallg
0.33	4606.16	15.49	800	0	FldFieldF::operator/=
0.28	4619.52	13.36	1600	0	fxdcp1borextrap
0.27	4632.1	12.58	201	0	cptimesteptur
0.25	4644.15	12.05	337885241	0	FldArrayF::operator[]
0.24	4655.6	11.45	1600	0	complrstate
0.2	4665.2	9.6	800	0	FldFieldF::add_and_assign
0.17	4673.48	8.28	20024	0	joincopyfld
0.15	4680.65	7.17	800	0	FldFieldF::mulsca_add_assign
0.14	4687.48	6.83	802	0	FldFieldF::mulsca_and_assign
4731.60					

FIG. A.2 – Profiling de la version CPU sur tesla

Profiling GPU Taille 4 Iter 200					
% time	cumulative seconds	self			name
		seconds	calls	s/call	
9.96	209.67	209.67	13609	0.02	memcpyHtoD
9.63	412.42	202.75	1603	0.13	operatordivf
8.61	593.62	181.2	9604	0.02	cons2tempf
7.43	750.06	156.44	3203	0.05	memcpyDtoH
6.68	890.55	140.49	2402	0.06	cons2speedf
5.92	1015.08	124.53	800	0.16	cons2prim
5.08	1121.99	106.91	2400	0.04	fdaddpart
4.64	1219.61	97.62	2400	0.04	kernel fxcroe5CU
4.21	1308.26	88.65	800	0.11	TmoSolverBase::computeLhs
2.96	1370.52	62.26	800	0.08	kernel compflux
2.56	1424.29	53.77	3207	0.02	kernel compgradsxaCU3d
2.53	1477.44	53.15	800	0.07	lhsupdate
2.23	1524.38	46.94	6400	0.01	compgradborder
2.05	1567.57	43.19	1600	0.03	compintfluxgradcorbdminusonetmp
2.03	1610.34	42.77	2401	0.02	compmu
1.8	1648.31	37.97	800	0.05	cons2primts
1.76	1685.39	37.08	800	0.05	FldFieldF::mulsca_mul_assign
1.65	1720.08	34.69	201	0.17	turcompmutwale
1.64	1754.61	34.53	2400	0.01	kernel fxccomp3oCU
1.54	1787	32.39	201	0.16	cpdtconv
1.53	1819.12	32.12	1600	0.02	kernel compbalancevec3
1.49	1850.49	31.37	6414	0	compbndgrad
1.04	1872.48	21.99	800	0.03	SouForcTerm::compute
1.03	1894.26	21.78	6406	0	extrapOf
0.99	1915.03	20.77	200	0.1	TmoSolverBase::solutionPEqualSolutionN
0.97	1935.38	20.35		#DIV/0!	spec_dexp2
0.91	1954.58	19.20	4632852509	0	FldFieldF::operator
0.74	1970.18	15.60	2002	0.01	isothwallg
0.74	1985.75	15.57	800	0.02	FldFieldF::operator/
0.65	1999.5	13.75	1600	0.01	fxdcp1borextraptmp
0.64	2012.9	13.40	337885241	0	FldArrayF::operator
0.6	2025.49	12.59	201	0.06	cpimesteptur
0.47	2035.39	9.90	800	0.01	FldFieldF::add_and_assign
0.4	2043.85	8.46	20024	0	joincopyfld
0.34	2050.98	7.13	800	0.01	FldFieldF::mulsca_add_assign
2.54	2104.4	53.42		#DIV/0!	(...)

FIG. A.3 – Profiling de la version GPU sur C1060

Profiling FERMI Taille 4 Iter 200					
%	cumulative time	self			name
		seconds	calls	s/call	
10.43	201.84	201.84	1603	0.13	operatordiv
9.93	393.93	192.09	12810	0.01	memcpyHtoD
9.32	574.26	180.33	9604	0.02	cons2tempf
8.39	736.5	162.24	3203	0.05	memcpyDtoH
7.26	876.93	140.43	2402	0.06	cons2speedf
6.43	1001.39	124.46	800	0.16	cons2prim
5.49	1107.65	106.26	2400	0.04	fdaddpart
4.57	1196.04	88.39	800	0.11	TmoSolverBase::computeLhs
2.77	1249.66	53.62	800	0.07	lhsupdate
2.39	1295.87	46.21	6400	0.01	compgradborder
2.35	1341.41	45.54	800	0.06	kernel_complflux
2.21	1384.25	42.84	2401	0.02	compmu
2.21	1426.95	42.7	1600	0.03	compintfluxgradcorrbdminusonetmp
2.1	1467.57	40.62	2400	0.02	kernel_fxcroe5CU
1.99	1505.98	38.41	800	0.05	cons2prints
1.93	1543.29	37.31	800	0.05	FldFieldF::mulsca_mul_assign
1.8	1578.12	34.83	201	0.17	turcompmutwale
1.79	1612.69	34.57	2400	0.01	kernel_fxccomp3oCU
1.67	1645.07	32.38	201	0.16	cpdtconv
1.66	1677.13	32.06	3207	0.01	kernel_comgradscaCU3d
1.62	1708.42	31.29		#DIV/0 !	compbndgrad
1.18	1731.2	22.78	1600	0.01	kernel_compbalancevec3
1.15	1753.5	22.3	800	0.03	SouForcTerm::compute
1.13	1775.31	21.81	6406	0	extrap0f
1.07	1795.97	20.66	200	0.1	TmoSolverBase::solutionPEqualSolutionN
1.04	1816.11	20.14		#DIV/0 !	spec_dexp2
1.01	1835.62	19.51	4632852509	0	FldFieldF::operator[]
0.8	1851.16	15.54	800	0.02	FldFieldF::operator/=
0.79	1866.45	15.29	2002	0.01	isothwallq
0.69	1879.78	13.33	1600	0.01	fxdcp1borextraptmp
0.65	1892.38	12.60	201	0.06	cptimesteptur
0.62	1904.3	11.92	337885241	0	FldArrayF::operator[]
0.52	1914.27	9.97	800	0.01	FldFieldF::add_and_assign
0.47	1923.28	9.01	20024	0	joincopyfld
0.37	1930.51	7.23	800	0.01	FldFieldF::mulsca_add_assign
0.2	1934.36	3.85		#DIV/0 !	(...)

FIG. A.4 – Profiling de la version GPU sur C2050

Annexe B

Tableau des temps

Taille du maillage	Octopus	Tesla	C1060	C2050
511485	541.118360	737.248071	346.589933	331.332974
1349985	1311.688177	1859.852684	876.603398	847.468037
2396485	2331.803704	3283.314423	1548.934445	1488.451746
3884985	3864.416918	5457.973118	2522.636761	2377.189538
5581485	5426.247961	7586.965509	3552.480300	

TAB. B.1 – Temps d'exécution d'elsA en secondes

taille du maillage	Octopus/C1060	Tesla/C1060	Octopus/C2050	Tesla/C2050
511485	1.5613	2.1271	1.6332	2.2251
1349985	1.4963	2.1217	1.5478	2.1956
2396485	1.5054	2.1197	1.5666	2.2059
3884985	1.5319	2.1636	1.6256	2.2960
5581485	1.5274	2.1357		

TAB. B.2 – Accélération d'elsA

Nombre de blocs	Octopus	Tesla	C1060	C2050
2	5446,35	7417,59	3738,19	3346,28
4	5420,14	7409,84	3719,89	3420,86
8	5540,20	7643,72	3835,89	3495,43
10	5616,07	7803,72	3895,77	3518,66
11	6684,52	7603,22	3884,47	3551,40
16	5548,05	7757,32	3970,34	3639,13
32	6082,65	7796,76	4178,25	3818,48
64	6012,12	8153,18	4662,97	4194,66

TAB. B.3 – Temps d'exécution d'elsA multi blocs sur un maillage de taille 61x329x261

Mesh size	Blocks number	CPU	GPU	Rapport
65x129x61	4	2,4411E+02	1,4744E+02	60,40%
65x129x61	8	2,4197E+02	1,5055E+02	62,22%
65x129x61	10	2,7658E+02	1,8662E+02	67,47%
65x129x61	11	2,6595E+02	1,7750E+02	66,74%
65x129x161	4	6,0660E+02	3,5554E+02	58,61%
65x129x161	8	6,1590E+02	3,6405E+02	59,11%
65x129x161	10	7,4739E+02	4,4785E+02	59,92%
65x129x161	11	6,9441E+02	4,3004E+02	61,93%
65x229x161	4	1,0840E+03	6,2686E+02	57,83%
65x229x161	8	1,0771E+03	6,2429E+02	57,96%
65x229x161	10	1,2886E+03	7,5356E+02	58,48%
65x229x161	11	1,1715E+03	7,0139E+02	59,87%
65x229x261	4	1,7377E+03	9,9767E+02	57,41%
65x229x261	8	1,7424E+03	1,0108E+03	58,01%
65x229x261	10	2,1271E+03	1,2338E+03	58,01%
65x229x261	11	1,9777E+03	1,1318E+03	57,23%
65x329x261	4	2,4975E+03	1,4109E+03	56,49%
65x329x261	8	2,5017E+03	1,4335E+03	57,30%
65x329x261	10	3,0090E+03	1,7291E+03	57,47%
65x329x261	11	2,7287E+03	1,5795E+03	57,88%
65x329x261	16	2,4783E+03	1,4424E+03	58,20%
65x329x261	32	2,6847E+03	1,4617E+03	54,44%
65x329x261	64	2,6573E+03	1,5501E+03	58,34%

TAB. B.4 – Temps d'exécution d'elsA sur 4 noeuds (1 GPU par noeud)