Université Pierre et Marie Curie

Rapport de Projet HPCA: Simulation de propagation d'ondes

Auteurs:
Amaury de Wargny
Tayyib Patel

Encadrants:
Pierre Fortin
Issam Said

17 Janvier 2016

Sommaire

1 Introduction

La résolution de problèmes dont le temps de calcul est élevé se trouve au centre des travaux en HPC. Depuis quelques années, de nombreux acteurs du secteur industriel du HPC ont étudié l'utilisation des GPU, des processeurs avec une structure hautement parallèle qui privilégient considérablement les cœurs de calcul par rapport aux CPU traditionnels.

L'objectif de ce projet est d'étudier la programmation GPU et le gain de temps qu'elle apporte, sur un problème concret de simulation de propagation d'ondes sismiques dans un domaine 3D modélisant une partie de l'écorce terrestre.

Pour cela, un programme nommé simwave, et son code séquentiel, représentant cette propagation nous a été fourni. Nous avons d'abord amélioré cette version sur un processus CPU en utilisant OpenMP, avant de la paralléliser sur un GPU en utilisant CUDA. Ensuite, nous finissons sur une parallélisation hybride CUDA+MPI.

2 Présentation de l'algorithme séquentiel

La simulation de propagation s'effectue sur une grille 3D de points. Une source crée l'onde qui sera ensuite propagée sur la grille sur plusieurs pas de temps, avec un traitement spécifique aux bords du domaine. Cette propagation est modélisée par la formule suivante en chaque point de la grille qui opère à chaque pas de temps :

$$U_{i,j,k}^{n+1} = \begin{cases} \frac{2}{\Gamma_{i,j,k}+1} U_{i,j,k}^{n} - \frac{\Gamma_{i,j,k}-1}{\Gamma_{i,j,k}+1} U_{i,j,k}^{n-1} + \frac{c_{i,j,k}^2 \Delta t^2}{\Gamma_{i,j,k}+1} \Delta U_{i,j,k}^{n} + c_{i,j,k}^2 \Delta t^2 s^n, & (i,j,k) \in \text{zone PML} \\ 2U_{i,j,k}^{n} - U_{i,j,k}^{n-1} + c_{i,j,k}^2 \Delta t^2 \Delta U_{i,j,k}^{n} + c_{i,j,k}^2 \Delta t^2 s^n, & (i,j,k) \in \text{sinon,} \end{cases}$$

avec notamment $U_{i,j,k}^{n}$ la valeur du champ d'onde au pas de temps n au point de la grille ($x = i\Delta x$, $y = j\Delta y$, $z = k\Delta z$).

Pour tester nos différentes implémentations, nous utiliserons principalement 3 cas:

- Cas 1: ./bin/simwave -v –grid 100,100,100 –nbsnap 10 –iter 100 –check
- Cas 2: ./bin/simwave -v –grid 400,400,400 –nosnap –iter 100 –check
- Cas 3: ./bin/simwave -v –grid 400,400,400 –nbsnap 10 –iter 100 –check

avec *grid* la taille de la grille en 3D, *nbsnap* la fréquence de sauvegarde de la grille et l'option *check* qui vérifie que les résultats sont corrects. Les temps d'exécution présentés à la suite dans ce rapport seront les meilleures temps obtenus sur 5 exécutions (les temps trop éloignés de la moyenne ne seront pas pris en compte). De plus, les temps sur GPU comprennent le temps d'initialisation et d'allocation mémoire sur le GPU.

-	V0	V0.5	V1	Accélération V0/V1
Cas 1	10.08	2.11	2.33	4.32
Cas 2	120.11	38.12	29.57	4.06
Cas 3	121.06	38.24	29.80	4.06

Figure 1 – Résultats sur Intel Core i7-4790

3 Premières améliorations

Nous commençons par paralléliser sur CPU à l'aide de OpenMP, dans le but de comparer les temps de calcul GPU avec des temps CPU accélérés. Par ailleurs, l'option check procède à des calculs sur CPU pour vérifier les résultats GPU. Il en devient ainsi nécessaire d'avoir un temps CPU accéléré.

Dans cette partie, les calculs sont effectués sur les ordinateurs de la PPTI, équipés de processeurs "Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz", eux-mêmes constitués de 4 cœurs physiques et 8 cœurs logiques.

3.1 Versions 0.5 et 1: Utilisation de OpenMP

La majorité des calculs se font dans une fonction nommée pwave_update_fields(), qui utilise trois boucles imbriquées pour chaque dimension de la grille. Dans cette version, nous avons parallélisé ces boucles à l'aide de directives pragma, sur la dimension z.

Par ailleurs, une condition se trouve à l'intérieur des boucles, ce qui résulte à beaucoup de cycles perdus. Ainsi, nous avons aussi ré-arrangé cette fonction afin d'éviter la condition. On nomme V0 la version séquentielle, V0.5 la version OpenMP de base et V1 la version OpenMP améliorée.

3.2 Résultats

Le tableau suivant résume les temps d'exécution, ainsi que les accélérations obtenues sur V0, V0.5 et V1, sur 8 threads.

L'accélération est sous-linéaire mais reste satisfaisante, et très utile pour l'option *check* dans la suite du projet.

4 Parallèlisation sur GPU en CUDA

Le but de cette partie est de déployer simwave sur un GPU en utilisant la librairie CUDA. Pour cela, deux GPU de modèle "Geforce GTX 480" (gpu1 et gpu2) et un GPU de modèle "Geforce GTX Titan" (gpu3) ont été mis à notre disposition.

4.1 V2: Version de base

Premièrement, nous avons codé une version basique, dans laquelle les données sont transférées au GPU. Celui-ci s'attèle à exécuter les calculs et rapatrie les données selon la valeur de *nbsnap*, et une

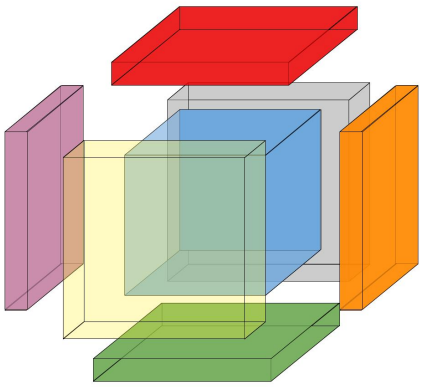

Figure 2 – V2.2

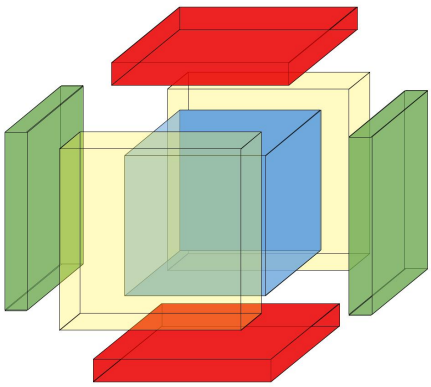

Figure 3 – V2.3

Une couleur représente l'espace calculé par un kernel. Le bleu correspond à la zone intérieure, les autres couleurs correspondent aux zones PML.

dernière fois après le dernier pas de temps.

4.2 V2.2 et V2.3: suppression de condition dans la boucle principale

Dans la boucle principale qui exécute les calculs, une condition est imposée qui décide si le point est dans une zone PML ou dans la zone intérieure de la grille (que nous avons retiré entre la V0.5 et la V1). Le but de la V2.2 et la V2.3 est de retirer cette condition.

La V2.2 utilise 7 kernels (fonction exécutée dans le GPU) pour calculer les données à chaque étape (bas, haut, gauche, droite, avant, arrière, milieu), illustré sur la figure 2.

Après quelques essais, nous remarquons que l'appel à plusieurs kernels est plus coûteux et avons ainsi décidé de réduire le nombre de kernels appelés à 4 (bas/haut, gauche/droite, avant/arrière, milieu), comme illustré sur la figure 3.

4.3 V2.4: utilisation de cudaStream

Une autre optimisation est l'utilisation de cudaStream. Ceux-ci sont des tampons mémoires particuliers qui envoient les données du GPU au CPU de façon non-bloquante. Ainsi, ils sont utiles lorsqu'il y a des *snapshots* à rapatrier, comme dans les cas 1 et 3. Lors du rapatriement des données, le GPU continue les calculs des itérations suivantes.

4.4 Résultats sur la V2

Nous avons rajouté les options –bx, –by et –bz dans le code afin de choisir les tailles de bloc lors de l'exécution. De plus, nous avons écrit un code en C qui détermine la taille de bloc optimale, utilisé avec l'option –tuning.

Par ailleurs, afin de présenter au mieux les gains en performance obtenus par la v2.4, nous avons effectué des tests en plus des trois cas présentés, en faisant varier le nombre de *snapshots*.

4.4.1 Résultats

Le GPU GTX 480 a une capacité limitée en mémoire par rapport au GPU GTX Titan. Les données étant contiguës sur l'axe x, puis l'axe y et l'axe z, nous n'avons pas pu prendre des tailles de blocs telles que $x > 32$, $y > 1$ et $z > 1$, sans qu'il n'y ait d'erreurs. Nous avons gardé la taille $(16, 1, 1)$ pour ce GPU. Sur le GPU GTX Titan, nous avons déterminé que la taille de bloc optimale était $(32, 4, 1)$.
Les tableaux suivants présentent les tests réalisés (OOM=Out Of Memory):

-	V1	V2	V2.3	V2.4	V1/V2.4
Cas 1	2.33	0.59	0.72	0.56	4.16
Cas 2	29.57	15.01	OOM	15.04	1.97
Cas 3	29.80	15.48	OOM	15.3	1.9

Figure 5 – Geforce GTX 480

-	V1	V2	V2.3	V2.4	V1/V2.4
Cas 1	2.33	0.67	0.74	0.64	3.64
Cas 2	29.57	5.32	11.09	4.96	5.96
Cas 3	29.80	5.78	11.55	5.17	5.76

Figure 6 – GeForce GTX Titan

La V2.4 étant une optimisation pour le rapatriement des données, nous l'avons essayé sur deux cas en plus, afin de comparer avec la V2:

- Cas 4 : ./bin/simwave -v –grid 400,400,400 –nbsnap 5 –iter 100 –check
- Cas 5 : ./bin/simwave -v –grid 400,400,400 –nbsnap 1 –iter 100 –check

Les tableaux ci-après présentent les résultats obtenus:

-	V2	V2.4	V2/V2.4
Cas 3	15.48	15.3	1.01
Cas 4	16.12	15.54	1.04
Cas 5	20.55	17.57	1.17

Figure 7 – Geforce GTX 480

-	V2	V2.4	V2/V2.4
Cas 3	5.78	5.17	1.12
Cas 4	6.26	5.32	1.18
Cas 5	10.03	6.67	1.50

Figure 8 – GeForce GTX Titan

4.5 Analyse des résultats

Sur le GPU GTX 480, on remarque une nette accélération dans le cas 1, mais qui diminue fortement sur les cas 2 et 3, où sont manipulées 4 fois plus de mémoire. En revanche, ceci ne pose pas de problème au GTX Titan, où l'accélération sur les cas 2 et 3 sont satisfaisantes.

En regardant les cas 4 et 5, nous voyons que l'accélération n'est pas très élevée entre la V2 et la V2.4, alors qu'il y a respectivement 2 fois et 10 fois plus de données rapatriées. On en déduit que le transfert GPU $\rightarrow$ CPU des résultats n'est pas coûteux.

Une autre optimisation que nous aurions pu faire est l'utilisation de mémoire partagée, où la différence avec la version de base aurait pu être plus marquée. Effectivement, le problème ici est *memory bound*, chaque calcul nécessitant de charger 24 données non-contiguës dans la grille.

5 Version 3: parallélisation multi-GPU

Dans cette partie, l'objectif est d'ajouter une parallélisation MPI au dessus de notre version C+CUDA de simwave, afin d'exploiter les trois GPU à notre disposition. Dans ce cadre, on ne considérera pas l'utilisation des *snapshots*.

5.1 V3: Version multi-GPU de base

On divise les points en N parties égales par rapport à l'axe z, à répartir sur N nœuds. Cependant, les nœuds sont inter-dépendants. Le calcul d'un point a besoin, d'une itération à l'autre, de la valeur du champ d'onde aux huit points l'avoisinant sur chaque axe. Ainsi, d'une itération à l'autre, chaque nœud devra envoyer et recevoir les données de la zone frontalière aux autres nœuds, tel qu'illustré sur la figure 10.

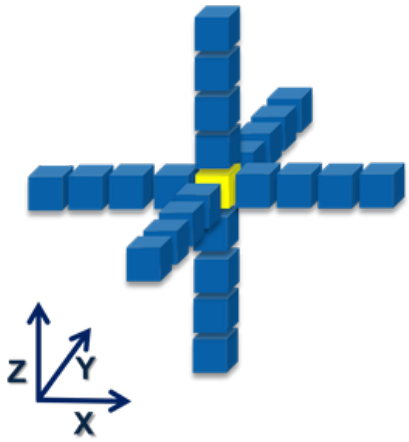

Figure 9 – *Chaque point a besoin de la valeur de 24 points avoisinants.*

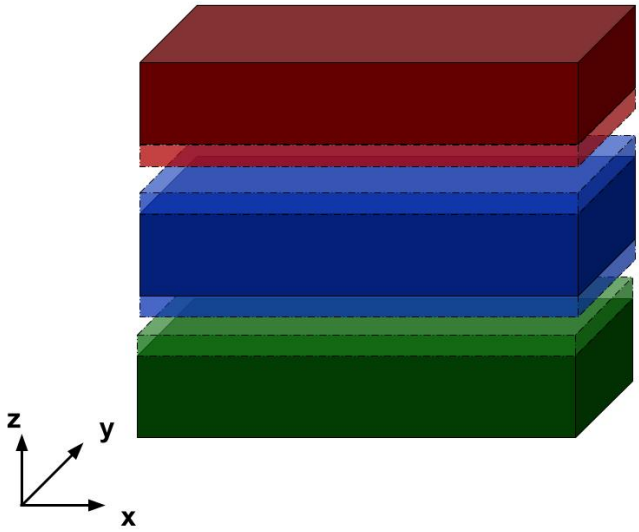

Figure 10 – *Chaque couleur correspond aux points traités par un nœud de calcul, les parties claires étant celles à envoyer au(x) nœud(s) voisin(s) à chaque itération.*

À chaque itération du calcul, il faut que chaque nœud:

1. exécute ses calculs respectifs.
2. récupère une partie des données, nécessaire aux autres nœuds, du GPU sur le CPU.

3. envoie et reçoit ces données en utilisant les directives *MPI_Send* et *MPI_Recv*.
4. envoie ces données du CPU au GPU.

Finalement, le nœud de rang 0 récupère toutes les données.

5.2 V3.1

La V3 de base possède un défaut majeur, chaque nœud alloue de l'espace pour l'ensemble total de la grille, alors qu'il n'a besoin en réalité que d'une partie de la grille. Ceci peut poser problème lorsqu'une grande taille de grille est nécessaire.

Notre but était ainsi d'allouer sur chaque nœud uniquement l'espace nécessaire plutôt que la grille en entière. Cependant, nous n'avons pas réussi à avoir des résultats cohérents, ce qui est dû aux changements des indices pour les différents tableaux manipulés, et nous avons abandonné cette version.

5.3 V3.2: utilisation de directives MPI asynchrones

L'envoi de données d'un nœud à l'autre étant potentiellement long, et sachant que seuls les calculs des zones frontalières nécessitent ces données, l'objectif de la V3.2 est d'utiliser les directives MPI asynchrones.

Ainsi, on reprend la V3 et on la modifie afin de:

1. exécuter les calculs.
2. envoyer les données nécessaires, en utilisant des directives MPI non-bloquantes.
3. exécuter les calculs des zones non-frontalières.
4. si besoin, attendre que les envois se terminent.
5. exécuter les calculs des zones frontalières.
6. boucler sur l'étape 2 tant qu'il reste des itérations.

5.3.1 Résultats sur la V3

Les résultats présentés ci-dessous sont sur les trois GPU à notre disposition. Le code exécuté étant le même pour les trois, la taille de bloc a été fixée à $(16, 1, 1)$ afin de s'adapter aux GPU GTX 480.

Par ailleurs, plus la dimension de la grille est grande, moins l'exécution sera stable. Nous avons ainsi décidé d'ajouter des cas. Ceux-ci sont tous sur 100 itérations, sans *snapshot* et avec des tailles de grille différentes:

- Cas 1*: 100,100,100
- Cas 2*: 200,200,200
- Cas 3*: 300,300,300
- Cas 4*: 400,400,400

On obtient ces résultats, en faisant varier le nombre de nœuds N:

-	V1	V3 (N=2)	V3.2 (N=2)	V3 (N=3)	V3.2 (N=3)
Cas 1*	2.29	1.33	1.19	1.62	1.35
Cas 2*	6.45	3.26	2.20	4.34	3.56
Cas 3*	15.11	7.68	4.41	9.40	7.44
Cas 4*	29.12	291.08	236.60	305.97	258.86

Figure 11 – Multi-GPU

5.4 Analyse des résultats

Plusieurs remarques:

1. L'accélération est très sous-linéaire et peu satisfaisante.
2. Pour une petite instance, l'utilisation de MPI va même ralentir le calcul.
3. Nous observons un gros défaut avec le cas 4*, sûrement dû à une mémoire à gérer trop importante.
4. L'exécution est plus lente lorsqu'il y a 3 nœuds plutôt que 2.

On en déduit que l'utilisation de MPI n'est pas compatible avec notre code GPU.

6 Conclusion

Une amélioration conséquente des temps d'exécution entre un code séquentiel et un code parallélisé avec des outils HPC est observable (voir figure 12 et 13).

Nous avons des temps satisfaisants obtenus par l'utilisation d'un GPU, tel que dans le cas 2, où l'accélération est d'un facteur ≈ 4 par rapport à un code optimisé utilisant OpenMP, et d'un facteur ≈ 24 sans OpenMP. En revanche, la parallélisation multi-GPU dans notre cas a été décevante.

Pour conclure, ce projet nous a permis de nous familiariser avec la programmation GPU en utilisant CUDA, de mieux comprendre le fonctionnement d'un GPU et d'en percevoir l'efficacité apparente. Dans de futurs travaux, des études plus approfondies avec des GPU homogènes pourraient révéler des résultats très satisfaisants.

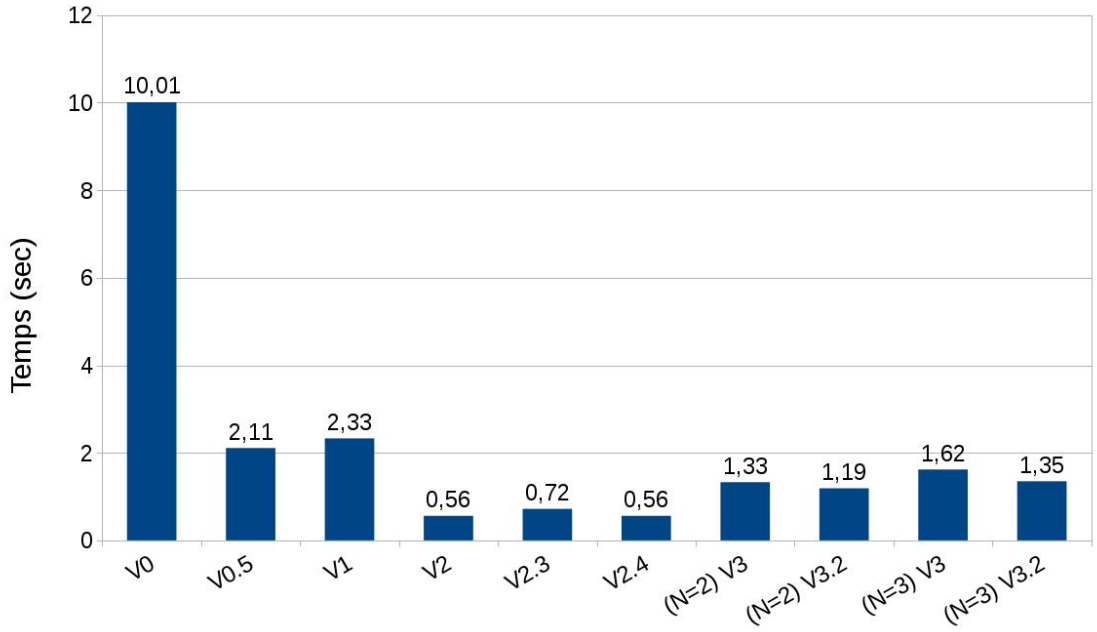

Figure 12 – *Temps d'exécution pour le cas 1*

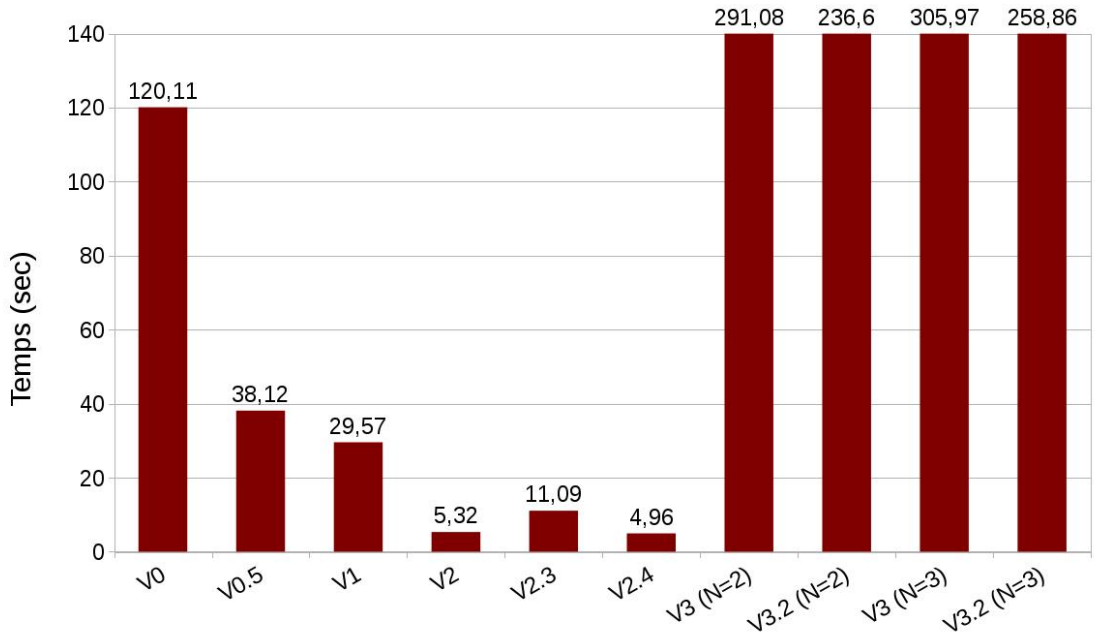

Figure 13 – *Temps d'exécution pour le cas 2*