Master 2 Informatique
Université de Bordeaux I

Projet d'**É**tude et de **D**éveloppement

Portage d'un code de lattice QCD sur GPU

Adrien Cassagne, Brice Mortier, Damien Pasqualinotto, Valentin Fréchaud

Encadré par Denis Barthou et Serge Chaumette
Sujet proposé par Denis Barthou

Bordeaux, le 27 mars 2013

Table des matières

I Le domaine . . . 1
I.1 QCD : Quantum chromodynamics . . . 1
I.2 L'algorithme étudié . . . 1
II L'existant . . . 2
II.1 Génération de code à partir de formules LaTeX . . . 2
II.2 Code généré . . . 2
II.3 QUDA . . . 3
III Portage sur GPU . . . 5
III.1 Objectifs . . . 5
III.2 Décomposition du travail . . . 5
III.3 Validité des résultats . . . 6
IV Optimisations . . . 7
IV.1 Optimisation des accès mémoire . . . 7
IV.2 Les opérations `xscal` et `xaxpy` . . . 7
IV.3 La réduction : `xnrm2` . . . 7
IV.4 Le coeur du calcul : `xgemm` . . . 8
IV.5 Traces d'exécution . . . 13
V Expérimentations . . . 15
V.1 Description des GPUs . . . 15
V.2 Conditions de tests . . . 15
V.3 *NVIDIA Quadro 4000* . . . 15
V.4 *NVIDIA GeForce GTX 660* . . . 16
V.5 *NVIDIA Tesla C2050* . . . 17
V.6 *AMD Radeon 7970* . . . 17
V.7 Bilan sur les résultats . . . 18
V.8 Pistes d'améliorations . . . 18
VI Conclusion . . . 19
Bibliographie . . . 20

Table des figures

1 Organisation originale du *buffer* `U` 2
2 Performances d'un calcul proche du *Conjugate Residual* avec QUDA . . . 4
3 Chronologie du projet en fonction des semaines 5
4 Multiplications matricielles dans un *workgroup* 10
5 Réorganisation de `U` (couples `uup/udn(dt/dx/dy/dz)` contiguës) 11
6 Accès à la mémoire partagée au sein d'un *workgroup* (`xgemm`) 12
7 Accès à la mémoire partagée après les transpositions dans `U` (`xgemm`) . . . 12
8 Accès à la mémoire partagée après les transpositions et les réarrangements dans `U` (`xgemm`) 12
9 Trace d'exécution de la version 1 13
10 Trace d'exécution de la version 2 14
11 Speedup par version - *NVIDIA Quadro 4000* 16
12 gigaflops/s par version - *NVIDIA Quadro 4000* 16
13 Speedup par version - *NVIDIA GeForce GTX 660* 16
14 gigaflops/s par version - *NVIDIA GeForce GTX 660* 16
15 Speedup par version - *NVIDIA Tesla C2050* 17
16 gigaflops/s par version - *NVIDIA Tesla C2050* 17
17 Speedup par version - *AMD Radeon 7970* 18
18 gigaflops/s par version - *AMD Radeon 7970* 18

I Le domaine

I.1 QCD : Quantum chromodynamics

La chromodynamique quantique est une théorie physique qui décrit l'intéraction des quarks et des gluons formant les particules comme les protons, neutrons ou mesons.

Simuler ces intéractions informatiquement est l'un des Grand Challenges de simulation. La méthode, appelée lattice QCD, consiste en la discrétisation de l'espace en une matrice à quatre dimensions appelée matrice de Dirac. De nombreuses équipes de recherche cherchent à calculer des matrices de plus en plus grosses afin de se rapprocher d'une taille de $256^3 \times 512$ qui permettrait une bonne discrétisation du modèle étudié.

Le coeur de cette simulation calcule $x = A^{-1}b$, A étant une matrice creuse. Pour ce faire, plusieurs méthodes sont possibles. Dans notre projet, nous avons porté la méthode *Conjugate Residual* (voir algorithme 1) sur GPU.

I.2 L'algorithme étudié

Dans l'algorithm 1, A est une matrice de complexes, b, x, r, Ap, p, Ar sont des vecteurs de complexes, $\alpha, \beta, rar, rarold$ sont des complexes et ϵ, n_r sont des réels.

Algorithm 1 Conjugate Residual [CR]

Ensure: $x = A^{-1} * b$
Require: diagonal(() $A - A\dagger$)
  $r \leftarrow b$
  $p \leftarrow r$
  $x \leftarrow 0$
  $n_r \leftarrow (r|r)$
  $Ap \leftarrow A * p$
  $rar \leftarrow (r|Ap)$
  while $(n_r > \epsilon)$ **do**
    $rarold \leftarrow rar$
    $\alpha \leftarrow rar\dagger / (Ap|Ap)$
    $x \leftarrow x + \alpha * p$
    $r \leftarrow r - \alpha * Ap$
    $n_r \leftarrow (r|r)$
    $Ar \leftarrow A * r$
    $rar \leftarrow (r|Ar)$
    $\beta \leftarrow rar/rarold$
    $p \leftarrow r + \beta * p$
    $Ap \leftarrow Ar + \beta * Ap$
  end while

L'algorithme *Conjugate Residual* 1 est une méthode itérative qui exhibe peu de parallélisme. Cependant, le produit $A \times r$ peut être entièrement parallélisé.

II L'existant

II.1 Génération de code à partir de formules LaTeX

QIRAL est un logiciel développé par l'équipe *Runtime*[1] de l'INRIA Bordeaux[2], dans le cadre du projet ANR PetaQCD[3] et plus particulièrement par M. Denis Barthou, professeur en informatique.

Ce dernier (QIRAL) est un langage de haut niveau permettant la résolution d'un problème *lattice QCD*. À partir d'une formule LaTeX, QIRAL génère du code C exploitant la puissance de calcul d'une machine multi-coeurs à l'aide de directives *OpenMP*.

II.2 Code généré

A Constantes

Voici les constantes propres au problème qui sont définies dans le programme :
- LX, LY, LZ, LT, les valeurs des quatre dimensions du problème d'entrée ;
- $L = LX \times LY \times LZ \times LT$, le volume total du problème d'entrée.

B Du parallélisme avec *OpenMP*

La parallélisation du code d'origine est effectuée à l'aide de simples directives *OpenMP* : `#pragma omp parallel for`.

C *Buffer* de déplacement `U`

Le *buffer* `U` contient les matrices de déplacement dans le lattice : c'est lui qui occupe le plus de place dans la mémoire. L'organisation du *buffer* `U` est présentée dans la figure 1.

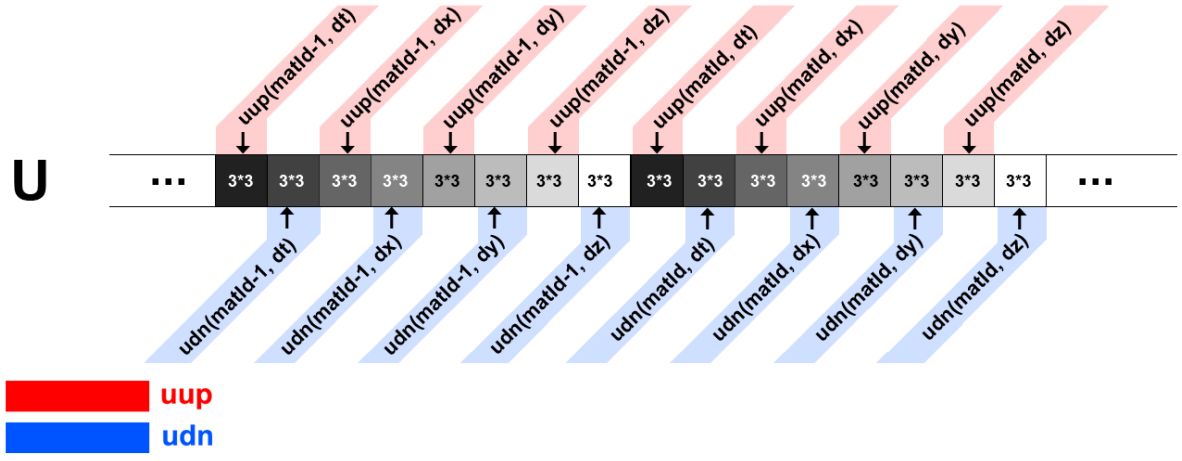

Figure 1 – Organisation originale du *buffer* `U`

1. Site de l'équipe *Runtime* : http://runtime.bordeaux.inria.fr/Runtime/
2. Site INRIA : http://www.inria.fr/centre/bordeaux
3. Site ANR PetaQCD : http://www.petaqcd.org

Cette organisation est cohérente avec le code 1 travaillant sur `U`. Elle permet une bonne localité des données.

```
#pragma for parallel
for (i = 0; i < L; ++i)
{
    complex tmp[12], tmp1[12], tmp2[12];

    xgemm(U[uup(iL, dt)], tmp, tmp1);
    xgemm(U[uup(iL, dx)], tmp, tmp1);
    xgemm(U[uup(iL, dy)], tmp, tmp1);
    xgemm(U[uup(iL, dz)], tmp, tmp1);

    xgemm(U[udn(iL, dt)], tmp, tmp2);
    xgemm(U[udn(iL, dx)], tmp, tmp2);
    xgemm(U[udn(iL, dy)], tmp, tmp2);
    xgemm(U[udn(iL, dz)], tmp, tmp2);
}
```

Listing 1 – Boucle de calcul principale

D *Buffers* inutiles

Dans la boucle de calcul utilisant le *buffer* de déplacements `U`, de nombreux *buffers* temporaires sont inutiles. Nous les avons supprimé et nous avons refactorisé le code.

E Dépassement de tableau

Il y a un dépassement de tableau inexpliqué dans le code d'origine (introduisant des erreurs de calcul). Celui-ci intervient dans la boucle de calcul présentée ci-avant (1).

Aucun patch n'a été fourni, et le client a été averti.

II.3 QUDA

QUDA [4] est une bibliothèque basée sur *CUDA*. Elle permet d'effectuer des calculs de lattice QCD sur les GPUs *NVIDIA*. Malheureusement nous n'avons pas eu le temps de tester cette bibliothéque pour la comparer avec nos travaux. Cependant, nous nous sommes appuyés sur un article [1] proche de notre projet afin d'avoir une idée sur les performances déjà atteintes dans ce domaine.

La figure 2 exprime le nombre de gigaflops par seconde obtenu pour un calcul proche du *Conjugate Residual* avec QUDA. Dans ce graphique, la courbe rouge représente le calcul en simple précision alors que la courbe noire correspond à l'exécution en double précision. Les performances sont mesurées en fonction de plusieurs GPUs (axe des abscisses). Puis le nombre de gigaflops/s est calculé pour un GPU (axe des ordonnées).

4. QUDA : http://lattice.github.com/quda/

Dans notre projet, nous avons uniquement travaillé sur des nombres en double précision. Si l'on considère que pour 48 GPUs les temps de communications sont négligeables, alors notre objectif est de se rapprocher au maximum de 30 gigaflops par seconde.

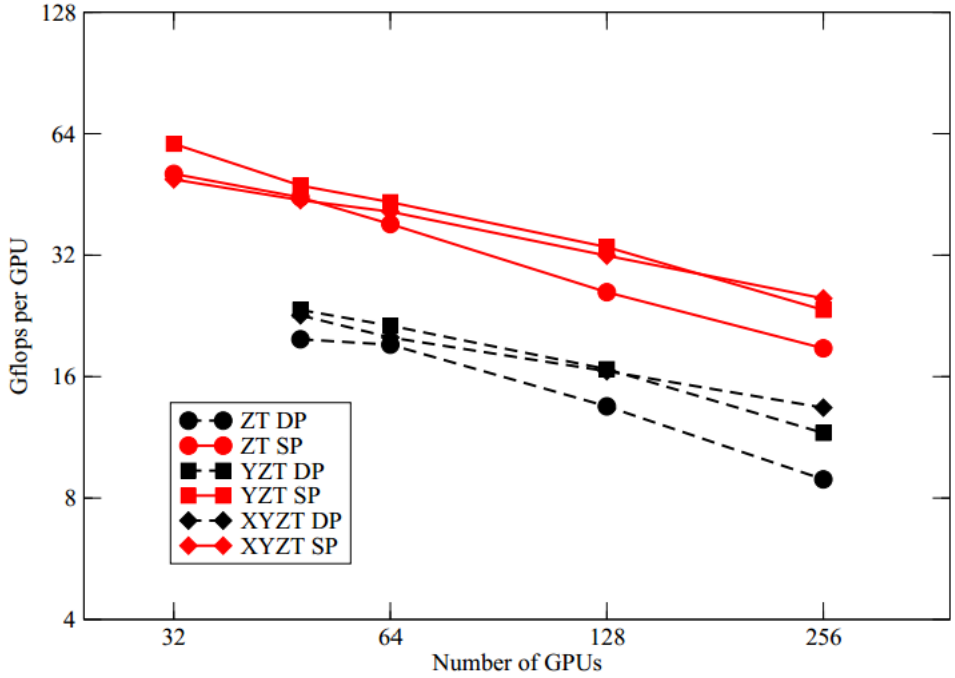

FIGURE 2 – Performances d'un calcul proche du *Conjugate Residual* avec QUDA

III Portage sur GPU

III.1 Objectifs

Avant de commencer le portage, nous avons essayé de déterminer les objectifs à atteindre. La vocation première de notre projet et d'améliorer le générateur de code QCD du client pour qu'il génère des codes plus rapides à l'aide du GPU. Après plusieurs discussions avec le client nous avons décidé de porter un code spécifique plutôt que de travailler directement sur le générateur. Ceci nous a permis de plus nous concentrer sur les optimisations en elles mêmes, plutôt que sur leurs intégrations au sein du générateur.

Porter et optimiser un code spécifique sur GPU permet d'aiguiller le client vers les optimisations intéressantes à générer.

III.2 Décomposition du travail

Dès le début du projet, nous avons fait le choix d'écrire le code GPU dans les deux langages les plus utilisés :

- *OpenCL* [2] de *Khronos Group* : projet libre fonctionnant sur bon nombre de GPUs ;
- *CUDA* [3] de *NVIDIA* : propriétaire et ne fonctionnant que sur les GPUs de la firme.

Ce choix a été motivé par notre volonté de comparer les deux APIs en terme d'utilisation et de performance. De plus *NVIDIA* propose un *profiler* de code[5] très interessant pour étudier l'efficacité des noyaux de calcul (code exécuté sur GPU, aussi appelé *kernels*).

Suite à cela, nous avons divisé notre projet en trois parties :

- le code CPU avec *OpenMP* ;
- le code GPU avec *OpenCL* ;
- le code GPU avec *CUDA*.

Dans notre arborescence de fichiers cela se traduit par trois dossiers distincts (*cpu*, *cuda* et *openCL*). Nous avons donc travaillé de manière séparé dans les différents "sous projets".

Dans un premier temps, nous avons itérativement porté l'opération `xaxpy`, suivie de `xnrm2`, puis `stencilCore` et enfin l'opération `xscal` (voir figure 3).

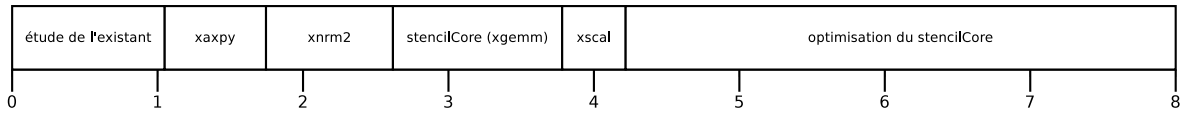

FIGURE 3 – Chronologie du projet en fonction des semaines

Une fois la totalité des opérations portées sur GPU, nous nous sommes affranchis de copies mémoires inutiles entre hôte (CPU) et périphérique (GPU). Cela a grandement simplifié le code. Nous nous sommes ensuite focalisés sur la partie du code qui prend le plus de temps : le `stencilCore`. Dans chacun des codes (*OpenCL* et *CUDA*), nous avons

5. *NVIDIA Visual Profiler* : https://developer.nvidia.com/nvidia-visual-profiler

décidé de créer un nouveau dossier pour chaque amélioration du `stencilCore`. Ce choix a entrainé une forte duplication de code mais nous a permis de comparer et travailler sur plusieurs versions en même temps.

III.3 Validité des résultats

À chaque portage d'une opération, nous avons vérifié l'exactitude des résultats en les comparant avec ceux de la version CPU originale. Cette méthodologie est relativement simple mais avec un peu de rigeur nous sommes arrivés à retrouver les bons résultats.

Lors du portage, nous avons étudié chaque opération dans le détail. Cette vision microscopique du code nous a permis d'assurer la similitude entre les opérations effectuées sur CPU et GPU.

IV Optimisations

IV.1 Optimisation des accès mémoire

Sur GPU, les accès à la mémoire sont souvent à l'origine des problèmes de performance. Pour une efficacité optimale, ils doivent être coalescés. En d'autres termes, chaque thread accède à une case mémoire qui lui est directement "dédiée".

Pour `tid` l'index d'un thread donné, un accès :
- `arr[tid]` est un coalescage parfait ;
- `arr[tid + 7]` est un bon coalescage, mais pose un problème d'alignement ;
- `arr[tid * 7]` n'est pas bon car il y a du padding entre deux accès ;
- `arr[random(0...N)]` est le pire cas possible.

IV.2 Les opérations `xscal` et `xaxpy`

`xscal` et `xaxpy` sont des opérations classiques des *BLAS 1*. `xscal` multiplie un vecteur par un scalaire : $v[N] = scal \times v[N]$ avec N la taille de v.

`xaxpy` ajoute un vecteur multiplié à un scalaire à un autre vecteur : $v2[N] = (scal \times v1[N]) + v2[N]$ avec N la taille de $v1$ et $v2$.

Nous avons essayé deux approches différentes pour porter ces opérations sur GPU, L est le nombre de vecteurs :
- chaque thread effectue un `xscal` ou `xaxpy` complet et on lance L threads ;
- chaque thread effectue un calcul d'un élément du vecteur et on lance $N \times L$ threads.

La deuxième version est à grain fin et les accès à la mémoire globale sont complètement coalescés. Nous avons retenu le deuxième choix car les performances sont bien meilleures.

IV.3 La réduction : `xnrm2`

Pour effectuer efficacement la réduction sur GPU, nous nous sommes appuyés sur un document officiel proposé par *NVIDIA* [6].

Sur GPU, il n'est pas possible de synchroniser des threads qui sont dans différents *workgroups* au sein d'un *kernel*. Pour réduire un tableau de grande taille il est pourtant nécessaire de synchroniser tous les threads entre eux. La solution consiste à lancer plusieurs fois le *kernel* de réduction, chaque retour sur l'hôte faisant office de barrière.

Afin d'accélérer la réduction, nous avons coalescé au maximum tous les accès mémoires.

6. Optimizing Parallel Reduction in CUDA [4]

IV.4 Le coeur du calcul : `xgemm`

A Présentation du noyau de calcul `stencilCore`

Le pseudo code 2 suivant est une simplification du *kernel* sur lequel nous apportons des optimisations.

```
1  constant complex Li1[4 * 4], Li2[4 * 4], Li3[4 * 4], Li4[4 * 4];
2  constant complex Li5[4 * 4], Li6[4 * 4], Li7[4 * 4], Li8[4 * 4];
3
4  kernel stencilCore(complex U[], complex matIn[], complex matOut[])
5  {
6    /* declaration des buffers temporaires */
7    complex tmp1[3 * 4];
8    complex tmp2[3 * 4];
9
10   /* stencil sur les 4 dimension dans le sens "up" */
11   xgemm34x44(&matIn[sup(matId, dx, bufMove)], Li1,  tmp1);
12   xgemm33x34(    &U[uup(matId, dx)         ], tmp1, tmp2);
13
14   xgemm34x44(&matIn[sup(matId, dy, bufMove)], Li2,  tmp1);
15   xgemm33x34(    &U[uup(matId, dy)         ], tmp1, tmp2);
16
17   xgemm34x44(&matIn[sup(matId, dz, bufMove)], Li3,  tmp1);
18   xgemm33x34(    &U[uup(matId, dz)         ], tmp1, tmp2);
19
20   xgemm34x44(&matIn[sup(matId, dt, bufMove)], Li4,  tmp1);
21   xgemm33x34(    &U[uup(matId, dt)         ], tmp1, tmp2);
22
23
24   /* stencil sur les 4 dimension dans le sens "down" */
25   xgemm34x44(&matIn[sdn(matId, dx, bufMove)], Li5,  tmp1);
26   xgemm33x34(    &U[udn(matId, dx)         ], tmp1, tmp2);
27
28   xgemm34x44(&matIn[sdn(matId, dy, bufMove)], Li6,  tmp1);
29   xgemm33x34(    &U[udn(matId, dy)         ], tmp1, tmp2);
30
31   xgemm34x44(&matIn[sdn(matId, dz, bufMove)], Li7,  tmp1);
32   xgemm33x34(    &U[udn(matId, dz)         ], tmp1, tmp2);
33
34   xgemm34x44(&matIn[sdn(matId, dt, bufMove)], Li8,  tmp1);
35   xgemm33x34(    &U[udn(matId, dt)         ], tmp1, tmp2);
36
37   [...]
38 }
```

Listing 2 – Pseudo code : `stencilCore`

Le pseudo code est volontairement simplifié au niveau des accès aux *buffers* `tmp1` et `tmp2`. `dx`, `dy`, `dz` et `dt` sont de simples constantes. `matId` est l'identifiant de position dans les matrices/*buffers*.

Les fonctions `sup`, `sdn`, `uup` et `udn` permettent d'obtenir les indices des éléments en fonction de la dimension et de la matrice sur laquelle on travaille.

Ce code est un stencil à 4 dimensions :
- les lignes $[08, 09]$ et $[22, 23]$ effectuent le stencil sur la dimension x ;
- les lignes $[11, 12]$ et $[25, 26]$ effectuent le stencil sur la dimension y ;
- les lignes $[14, 15]$ et $[28, 29]$ effectuent le stencil sur la dimension z ;
- les lignes $[17, 18]$ et $[31, 32]$ effectuent le stencil sur la dimension t.

Nous reviendrons et nous nous appuierons sur ce pseudo code (2) dans les explications et optimisations suivantes.

B Version 1.0 : gros grain de calcul

Dans cette version, chaque threads exécute les `xgemm` entièrement. Afin de traiter la totalité du calcul, le *kernel* `stencilCore` est lancé avec L threads. Ce mode de fonctionnement est calqué sur la version *OpenMP* originale.

Le `stencilCore` de cette version ne dépend pas de la taille des *workgroups*. Nous avons donc choisi de le lancer avec des *workgroups* de taille 256 (c'est la valeur qui donne les meilleurs résultats sur la majorité des architectures).

Les performances globales obtenues ne sont pas bonnes.

C Version 1.1 : faible grain de calcul

Comme nous l'avons déjà expliqué avec les opérations `xaxpy` et `xscal`, le choix d'un grain plus fin amène souvent un meilleur coalescage des accès aux données et de meilleures performances.

Dans cette version, chaque thread n'exécute qu'une partie des `xgemm`. Il faut $3 \times 4 = 12$ threads pour calculer un `xgemm` complet. En effet, chaque thread calcule un unique élément de la matrice résultat. Il faut donc $12 \times L$ threads pour effectuer le calcul complet.

Comme le montre le code, le résultat de chaque `xgemm` doit être stocké dans un *buffer* temporaire. Il faut donc que les 12 threads qui s'occupent d'un même `xgemm` partagent les variables `tmp`. Nous lançons donc le `stencilCore` de cette version avec L *workgroups* de 12 threads chacun et les *buffers* `tmp` sont stockés en mémoire partagée.

D Version 1.2 : optimisation de l'organisation des threads

Dans la version précèdente, le `stencilCore` fonctionne avec des *workgroups* composés de 12 threads. Le problème c'est qu'au niveau matériel les GPUs sont prévus pour être les plus performant en lancant des *workgroups* de taille multiple de 16. Dans la version 1.1, le programme lance des *workgroups* de 16 threads et 4 de ces threads ne font rien (ceci est géré automatiquement par *OpenCL* et écrit explicitement dans la version *CUDA*).

L'objectif de cette version est d'éviter ce gaspillage de threads. Le matériel nous impose de lancer des *worgroups* avec un nombre de threads multiple de 16. De plus, suite

à l'optimisation précèdente, il faut 12 threads pour calculer un `xgemm` complet et stocker son résultat dans un *buffer* temporaire. Ces 12 threads doivent être dans un même *workgroup* pour pouvoir stocker les *buffers* temporaires en mémoire partagée.

Cette optimisation propose donc de lancer des *workgroups* avec un nombre de threads multiple à la fois de **12** et de **16**. Pour ce faire, nous lançons les *workgroups* de la façon suivante :

– $WorkgroupSize = 12 \times \textbf{\textit{multiple}}$

En choisissant ***multiple*** de façon à ce que :

– $WorkgroupSize \mod 12 = 0$;

– $WorkgroupSize \mod 16 = 0$.

La valeur de ***multiple*** a été décidée en fonction de l'architecture matérielle :

– les cartes *Fermi* possèdent 16 unités de calculs sur les flottants en double précision (par *streaming multiprocessor*) : $multiple = 16$;

– les cartes *Kepler* 3.0 possèdent 8 unités de calculs sur les flottants en double précision (par *streaming multiprocessor*) : $multiple = 8$;

– les cartes *Kepler* 3.5 possèdent 64 unités de calculs sur les flottants en double précision (par *streaming multiprocessor*) : $multiple = 64$.

Ainsi, chaque *workgroups* s'occupe d'effectuer plusieurs `xgemm` en même temps. Par exemple sur les cartes *Fremi*, chaque *workgroups* effectue 16 `xgemm` en même temps. Notons que pour cette version les *workgroups* stockent plusieurs fois chacun des *buffers* temporaires (*multiple* fois).

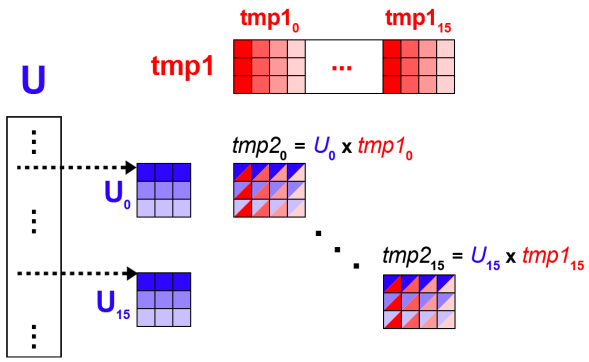

FIGURE 4 – Multiplications matricielles dans un *workgroup*

Grâce à cette optimisation nous évitons de perdre des unités de calcul.

E Version 1.3 : simplification des calculs

Nous avons remarqué que certains des `xgemm` étaient effectués avec des matrices creuses et constantes (*buffers* de type `Li*`). Dans cette version nous avons écrit "en dur" une multiplication spécifique à chaque matrice creuse afin de diminuer le nombre de calculs sur les flottants.

Cette optimisation a presque divisé le nombre d'opérations flottantes par deux.

F Version 1.4 : supression d'accès mémoire inutiles

Dans le *kernel* `stencilCore` (2), les accès à la matrice `matIn` dépendent d'un tableau de déplacement (`bufMove`) : `matIn[sup(matId, dx, bufMove)]`.

La fonction `sup` a besoin du *buffer* `bufMove`. Nous avons supprimé ce *buffer* au profit de plus de calcul dans la fonction `sup` : `matIn[sup(matId, dx)]`. Nous avons fait la même chose pour la fonction `sdn`.

G Version 1.5 : suppression de barrières

Nous avons volontairement omis de parler des barrières de synchronisation dans le pseudo code GPU (2). Dans le code réel du `stencilCore` il y a une barrière entre chaque groupe de deux `xgemm`. Nous avons remarqué que l'ordre des `xgemm` avait un impact sur la necessité des barrières. Dans la version 1.5, nous avons changé l'ordre des `xgemm` afin de minimiser le nombre de barrières : nous sommes passés de 17 à 13 barrières.

H Version 1.6 : réorganisation du *buffer* `U`

Dans cette version nous avons réfléchi à la façon dont le tableau `U` était accédé lors des précédentes versions. Lorsque le *kernel* `stencilCore` (2) est lancé sur le *buffer* `U` (figure 1), nous constatons qu'entre deux accès aux 9 éléments des matrices `uup(matId -1, dt)` et `uup(matId, dt)` il y a un saut de 9×8 éléments (du début de la matrice pointée par `udn(matId -1, dt)` à la fin matrice pointée par `udn(matId -1, dz)`). Il y a donc un problème d'alignement parce qu'un *workgroup* traite plusieurs matrices en même temps ($nombreDeMatrices = multiple$).

La figure 5 montre la réorganisation de `U`.

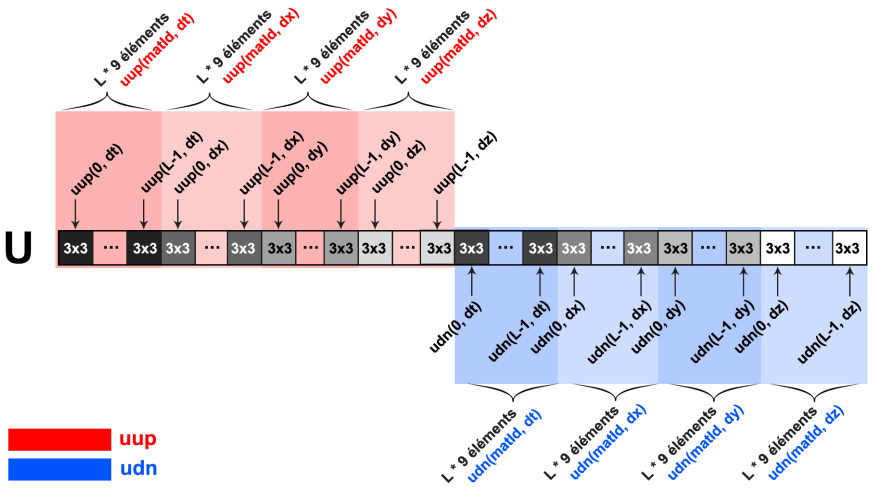

FIGURE 5 – Réorganisation de `U` (couples `uup/udn(dt/dx/dy/dz)` contiguës)

Lorsque certains threads d'un *workgroup* travaillent sur la matrice `matId` et que d'autres threads de ce même *workgroup* travaillent sur la matrice `matId +1` alors les matrices `matId` et `matId +1` sont maintenant contiguës en mémoire.

Nous avons ensuite essayé de mieux coalescer les accès dans la mémoire partagée. La figure 6 présente les accès mémoires originaux (en haut on retrouve les threads et en bas la mémoire partagée).

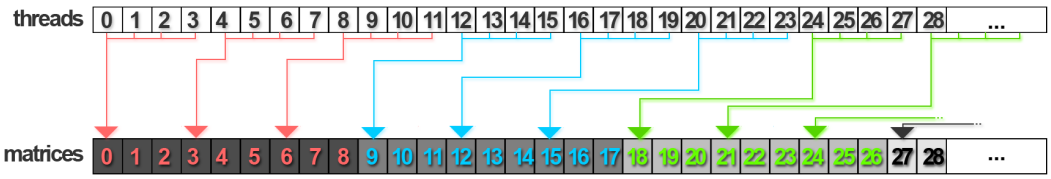

FIGURE 6 – Accès à la mémoire partagée au sein d'un *workgroup* (`xgemm`)

En transposant les matrices de `U`, nous obtenons un meilleur coalescage (voir sur la figure 7).

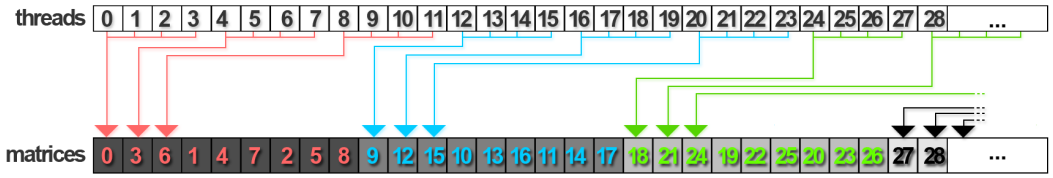

FIGURE 7 – Accès à la mémoire partagée après les transpositions dans `U` (`xgemm`)

Sur la figure 7, nous constatons que le coalescage est meilleur que sur la figure 6. Cependant nous remarquons aussi que l'on peut mieux faire. La figure 8 expose un nouveau réarrangement des matrices.

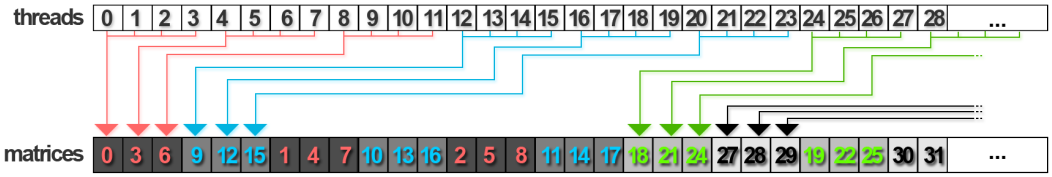

FIGURE 8 – Accès à la mémoire partagée après les transpositions et les réarrangements dans `U` (`xgemm`)

Ce nouveau réarrangement (figure 8) propose de regrouper les colonnes des matrices contiguës deux à deux pour profiter d'un coalescage encore plus efficace.

Version 1.6.1 : compatibilité avec *Kepler 3.5* En copiant la matrice `U` en mémoire partagée, nous dépassons la limite autorisé (48Ko) dans le cas où *multiple* = 64. De plus, nous avons constaté que sur l'architecture *Kepler 3.0*, l'impact de la mise en mémoire partagé de `U` est négligeable. Cette version propose de garder le *buffer* `U` en mémoire globale et rend donc possible son exécution sur une architecutre de type *Kepler 3.5* avec *multiple* = 64. Nous n'avons malheureusement pas pu tester nos *kernels* sur une architecture aussi récente.

I Version 2.0 : une autre vision

Dans la version 2 nous avons revu l'architecture générale du `stencilCore`. Au lieu de faire tous les appels aux `xgemm` dans un seul *kernel* `stencilCore`, nous avons créé un *kernel* spécifique pour chaque `xgemm` (un *kernel* `xgemm33x34` et un *kernel* `xgemm34x44`).

Cette approche a l'avantage d'être plus décomposée au niveau du code de l'hôte. Cependant les performances sont un peu moins intéressantes et nous ne l'avons pas aussi bien optimisée que la version 1.

IV.5 Traces d'exécution

La figure 9 est une trace d'exécution de la version 1.2 sur GPU (1 itération). Cette version est relativement claire et nous permet de d'identifier les *kernels* simplement. On remarque que la majorité du temps est passée dans le `stencilCore` (75%). Cela justifie le temps que nous avons consacré à son optimisation.

Les autres *kernels* (`xMult`, `xMultAdd`, `reduction`, etc.) sont secondaires. Les *kernels* `xMult` permettent de calculer les `xscal` et les *kernels* `xMultAdd` permettent de calculer les `xaxpy` (voir section IV.2).

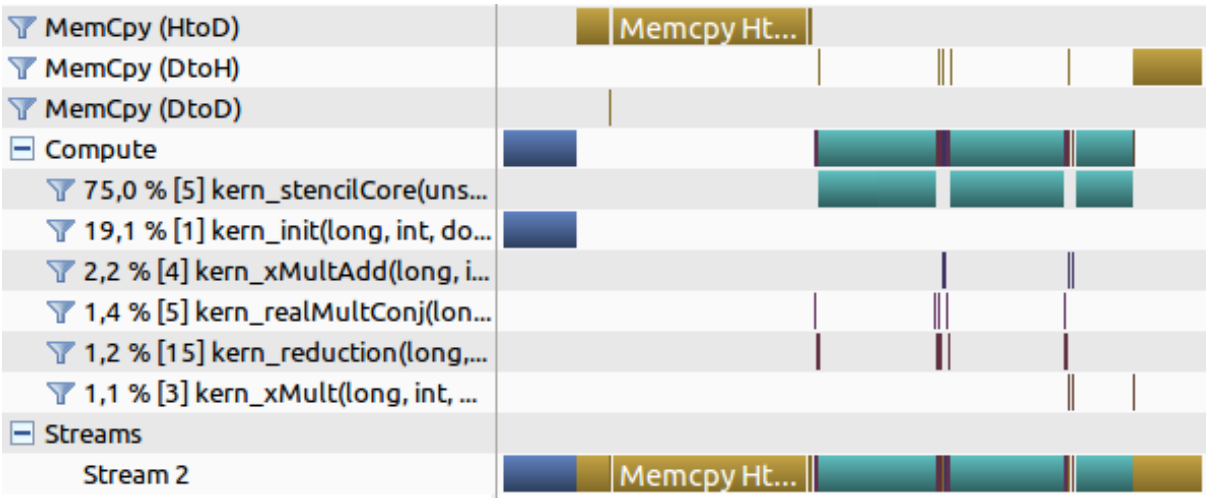

FIGURE 9 – Trace d'exécution de la version 1

La figure 10 est une trace d'exécution de la version 2.0 sur GPU (1 itération). On remarque qu'il y a beaucoup plus de *kernels* lancés. C'est normal puisque nous avons

décomposé le `stencilCore` en plusieurs sous `kernels` (`xgemm33x34` et `xgemm34x44`).

Cette décomposition permet d'analyser plus finement les temps d'occupation au sein du `stencilCore`. Ainsi, sans surprise, on remarque que le temps d'exécution des *kernels* `xgemm34x44` est plus grand que celui des *kernels* `xgemm33x34` (42,2% contre 35,5%).

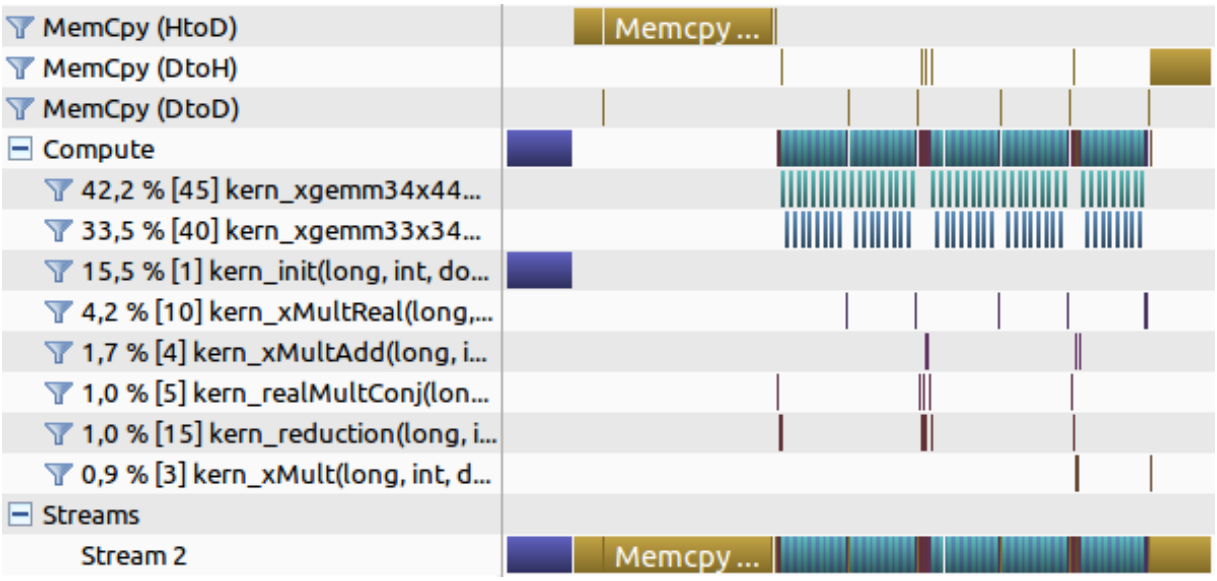

FIGURE 10 – Trace d'exécution de la version 2

V Expérimentations

V.1 Description des GPUs

Le tableau 1 récapitule les caratéristiques principales des 4 GPUs avec lesquels nous avons testé notre application.

Modèle	Unités de calcul	gigaflops/s	Mémoire	Fréquence mémoire
NVIDIA GTX 660	40 @ 1032 Mhz	78,0	2048 Mo	1502 Mhz
NVIDIA Quadro 4000	128 @ 950 Mhz	243,2	2048 Mo	700 Mhz
NVIDIA Tesla C2050	224 @ 1150 Mhz	515,0	3072 Mo	1500 Mhz
AMD Radeon 7970	512 @ 925 Mhz	947,2	6144 Mo	1375 Mhz

TABLE 1 – Caractéristiques des GPUs de test

Le nombre d'unités de calcul ainsi que les gigaflops/s sont relatifs aux calculs doubles précisions. Les GPUs *NVIDIA Quadro 4000* et *NVIDIA Tesla C2050* sont basés sur l'architecture *Fermi*[7] alors que le GPU *NVIDIA GTX 660* est basé sur l'architecture *Kepler*[8] (plus récente).

V.2 Conditions de tests

Pour chacuns de nos tests, nous avons lancé l'exécution du programme sur 143 itérations avec $L = 16 \times 16 \times 16 \times 32$. Cette taille de problème représente environ 400 Mo dans la mémoire. Nous avons essayé avec une taille plus importante ($L = 24^3 \times 48$) mais dans ce cas la version 2.0 ne fonctionne plus à cause d'un manque de mémoire GPU (plus de 2048 Mo).

Nous mesurons le speedup par rapport à la version *OpenMP* originale qui est exécuté sur une processeur *Intel Xeon W3600 @ 3,47Ghz* (6 coeurs avec la technologie *Hyper-Threading* activée). Le temps séquentiel sur CPU est de 51,002 secondes.

Les diagrammes de gigaflops par seconde sont calculés à partir du nombre d'opérations flottantes contenues dans le `stencilCore`. Ce nombre d'opérations est recalculé pour chaque version.

V.3 *NVIDIA Quadro 4000*

La version 1.1 est bien plus efficace que la version 1.0 même si l'on n'utilise pas toutes les unités de calcul disponibles. Le passage à un grain plus fin améliore les performances significativement. La version 1.2 apporte un réel bénéfice par rapport à la version 1.1. L'utilisation de toutes les unités de calcul disponibles est primordiale. De plus, dans cette version, nous sollicitons beaucoup plus les accès à la mémoire partagée.

7. *Fermi* : http://fr.wikipedia.org/wiki/Fermi_(architecture_de_carte_graphique)
8. *Kepler* : http://fr.wikipedia.org/wiki/Kepler_(architecture_de_carte_graphique)

La version 1.3 divise le nombre d'opérations flottantes par 2 (environ) cependant le speedup n'est pas doublé. Cette version reste néanmoins plus performante que la version 1.2. Le gain apporté par les versions 1.4 et 1.5 est anecdotique.

La version 1.6 avec la réorganisation du *buffer* `U` apporte aussi un gain considérable sur l'architecture *Fermi*.

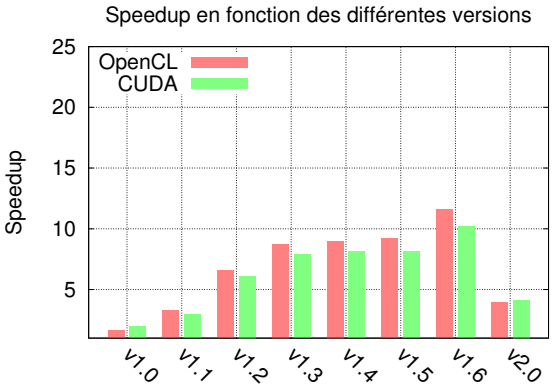

FIGURE 11 – Speedup par version - *NVIDIA Quadro 4000*

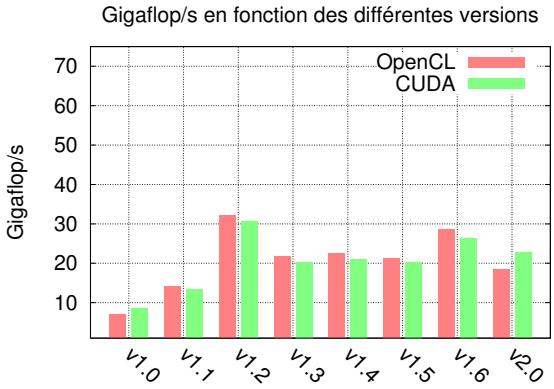

FIGURE 12 – gigaflops/s par version - *NVIDIA Quadro 4000*

La figure 12 exprime le nombre de gigaflops par seconde. On remarque que c'est dans la version 1.2 que nous effectuons le plus d'opérations flottantes par seconde. Quand on diminue de nombre de calcul (version 1.3), le temps ne diminue pas proportionnellement et donc le nombre d'opérations flottantes par seconde diminue. Le pic théorique de la *NVIDIA Quadro 4000* est de 243,2 gigaflops/s en double précision. Nous sommes très loin d'exploiter pleinement la carte.

V.4 *NVIDIA GeForce GTX 660*

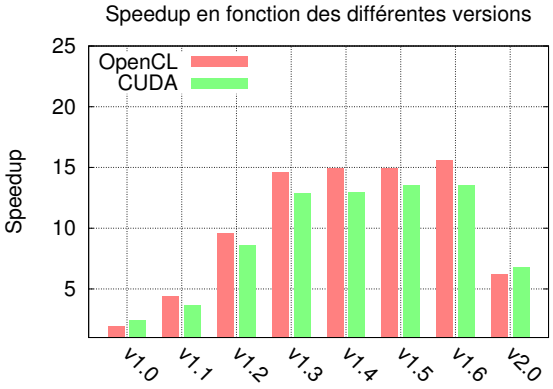

FIGURE 13 – Speedup par version - *NVIDIA GeForce GTX 660*

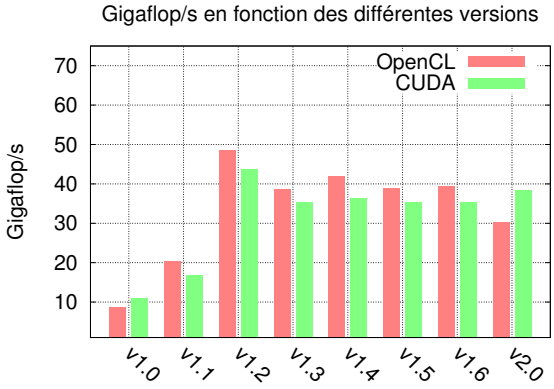

FIGURE 14 – gigaflops/s par version - *NVIDIA GeForce GTX 660*

Sur une architecture de type *Kepler 3.0*, le gain apporté par les versions 1.1, 1.2 et 1.3 est très net. La diminution du nombre de calculs flottants (version 1.3) a plus d'effet que

sur *Fermi*. Cependant le gain des versions 1.4, 1.5 et 1.6 est marginal. La réorganisation de `U` n'apporte pas le même gain que sur *Fermi*.

Notons que la puissance de calcul théorique de la *GeForce GTX 660* est très inférieure à celle de la *Quadro 4000* : 40 unités de calcul doubles précisions contre 128. Pourtant nous avons de meilleures accélérations avec la *GeForce GTX 660*. Le pic théorique de la *GeForce GTX 660* est de 78 gigaflops/s en double précision. Le nombre d'opérations flottantes en version 1.2 est supérieur à la moitié du pic (figure 14). L'architecture *Kepler* s'adapte mieux à notre programme que l'architecture *Fermi*.

Nous pensons que cela est dû à la fréquence de la mémoire qui est de 1500 Mhz pour la *GeForce GTX 660* contre 700 Mhz pour la *Quadro 4000*. Notre problème est limité par les accès mémoires parce que nous ne réutilisons pas beaucoup les données (multiplications matricielles de petites matrices).

V.5 *NVIDIA Tesla C2050*

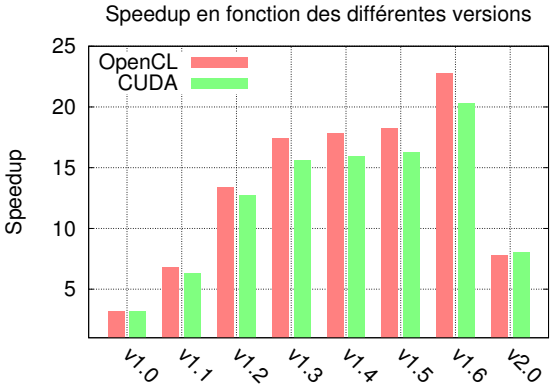

FIGURE 15 – Speedup par version - *NVIDIA Tesla C2050*

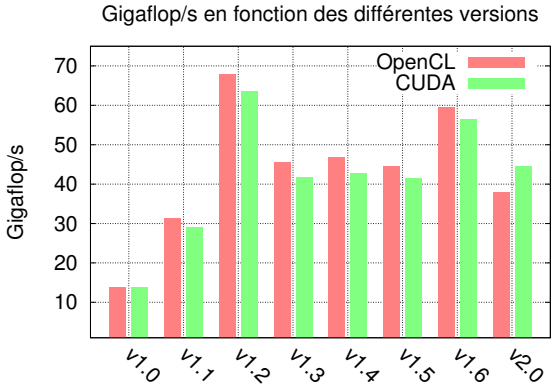

FIGURE 16 – gigaflops/s par version - *NVIDIA Tesla C2050*

La *Tesla C2050* est une carte dédiée au calcul scientifique. Son architecture est proche de celle de la *Quadro 4000* cependant la fréquence de la mémoire est nettement supérieure.

Sur la figure 15 on peut voir que le speedup s'améliore grandement ($\times 2$ par rapport à la figure 11). Les améliorations du speedup en fonction des versions sont identiques à la *Quadro 4000*. Cela confirme bien la similitude des architectures (*Fermi*).

Le nombre de gigaflops/s (figure 16) est très faible compte tenu de la puissance théorique maximale de la carte (68 gigaflops/s contre un maximum de 515 gigaflops/s).

V.6 *AMD Radeon 7970*

La *Radeon 7970* est une carte de dernière génération normalement dédiée aux jeux vidéo. Dans les figures 17 et 18 nous n'avons pas pû comparer *OpenCL* et *CUDA* car

CUDA ne fonctionne que sur les cartes *NVIDIA*. De plus, la version 2.0 n'a pas pu être mesurée suite à un problème de lancement non résolu sur cette carte.

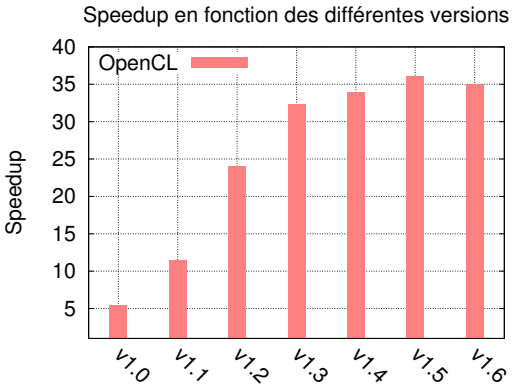

FIGURE 17 – Speedup par version - *AMD Radeon 7970*

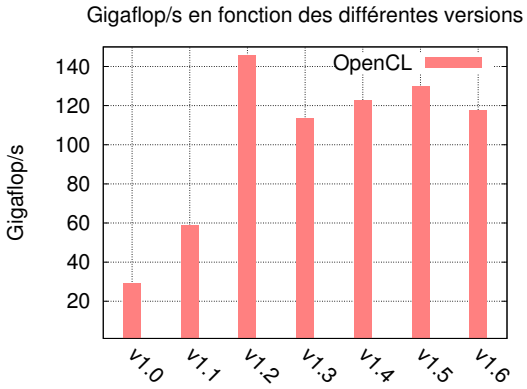

FIGURE 18 – gigaflops/s par version - *AMD Radeon 7970*

Sur la figure 17 on remarque que l'évolution du speedup en fonction des versions est assez similaire à la *GeForce GTX 660* (figure 13). Cependant le speedup maximal est nettement plus élevé qu'avec les autres cartes (speedup de 36 en version 1.5). De plus, on remarque aussi que la réorganisation du *buffer* `U` pénalise les performances. Nous pensons que cela est dû aux mauvaises performances de la mémoire partagée sur la *Radeon 7970*.

Les performances en gigaflops/s (figure 18) sont impressionnantes par rapport aux autres GPUs (145 gigaflops/s). Cependant on reste très loin du pic de la carte qui est de 947,2 gigaflops/s.

V.7 Bilan sur les résultats

Comme exposé dans les résultats pour chaque GPUs, nous n'exploitons qu'environ 15% des gigaflops/s potentiels. Cependant, ceci reste cohérent par rapport aux travaux déjà effectués dans ce domaine (2).

V.8 Pistes d'améliorations

A Multi-GPU

Nos versions du code n'ont pas été conçues pour fonctionner directement sur plusieurs GPUs. Cependant nous pensons que cela serait possible avec une simple répartition du travail. Cette amélioration permettrait d'accroître sérieusement les performances.

B Version 2.0

Afin de comparer les versions 1.x avec la version 2.0, il aurait été intéressant de pousser nos expérimentations plus loin sur la version 2.0. En effet, les diagrammes 12, 14, 16 et 18 montrent que cette version sollicite bien la carte.

VI Conclusion

Ce projet a été pour nous l'occasion d'apprendre et/ou de mieux comprendre le monde du calcul sur GPU. Nous avons porté le code original dans les deux grands langages existants (*OpenCL* et *CUDA)*. Cela nous a donné du recul sur les avantages et les inconvénients de l'un et de l'autre. Pour résumer, *OpenCL* est une bonne abstraction du matériel alors que *CUDA* est plus bas niveau mais son API est plus agréable à utiliser.

Les performances que nous avons obtenues sont satisfaisantes bien que perfectibles. Dans tous les cas nous avons un speedup de l'ordre de 10 par rapport au CPU. Cette accélération montre l'intérêt d'utiliser un GPU pour ce type de code. De plus, les nouvelles architectures (*Kepler*) sont beaucoup plus efficaces sur le code que nous avons implémenté.

Cependant le développement sur GPU est ardu. Il nécessite l'utilisation d'au moins deux langages : un langage hôte (*C*, *C++*, *Java*) et une API (*CUDA* ou *OpenCL*). De plus, il y a très peu d'outils pour debugger (même si *NVIDIA* et *CUDA* progressent beaucoup dans ce domaine).

Intel propose aujourd'hui son nouvel accélérateur : le *Xeon Phi*. Ce dernier est basé sur une architecture *x86* ; et *Intel* promet que le développement est plus simple que sur GPU. Il serait donc intéressant de comparer les performances du *Xeon Phi* avec celles des GPUs classiques.

En définitive, nous retirons une expérience très positive de ce projet. Le défis proposé est intéressant. Il est proche de la recherche et encore peu exploré : cela nous a beaucoup motivé. Nous avons aimé travailler en groupe et pensons avoir réussi à collaborer efficacement.

Bibliographie

[1] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice qcd beyond 100 gpus. In *2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, sept. 2011.

[2] Opencl. http://www.khronos.org/.

[3] Cuda. https://developer.nvidia.com/category/zone/cuda-zone.

[4] Mark Harris. Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.