Université Paris 7 Denis Diderot
UFR d'Informatique

Stage de fin d'études en vue de l'obtention du diplôme de
Master 2 Ingénierie Informatique parcours SRI

Calcul Haute Performance
sur **Carte Graphique**
Accélération d'algorithmes de traitement d'image avec **CUDA**

Antoine Pédron
<antoine.pedron@gmail.com>

Stage Effectué du 1^{er} Avril au 30 Septembre 2008

Responsables de stage:
Pierre Kestener, Yassir Moudden

Lieu du stage:
Commissariat à l'**E**nergie **A**tomique au Cente de Saclay
à la **D**irection des **S**ciences de la **M**atière
à l'**I**nstitut de **R**echeche sur les Lois **F**ondamentales de l'**U**nivers

9 septembre 2008

Remerciements

Je tiens à remercier toutes les personnes du département de l'IRFU du CEA Saclay pour leur accueil et plus particulièrement M.Michel Mur chef du SEDI, ainsi que M. Denis Calvet chef du TRAPS, de m'avoir permis d'effectuer mon stage au sein de leur département.

Je remercie beaucoup M. Pierre Kestener et M. Yassir Moudden, mes deux responsables de stage, pour leur disponibilité et leur soutien. Leur aide précieuse m'a permis de réaliser mon travail dans les meilleures conditions.

Je remercie Mlle Yncia Fidaali pour son positivisme à l'égard de mon travail ainsi que de sa présence chaleureuse.

Je tiens à aussi à remercier Mme Valérie Gautard, M.Bruno Thooris, M.Daniel Pomareède et M.Frederic Chateau.

Enfin, j'adresse mes remerciements à l'ensemble de l'équipe enseignante et d'administration de l'UFR Informatique de l'université Paris VII pour mes cinq années de formation. La diversité de l'ensemble des matières enseignées ainsi que leur qualité m'ont donné une culture qui m'a été précieuse durant mon stage, et qui le sera aussi à l'avenir.

..

Table des matières

Introduction **5**

1 Le Commissariat à l'Energie Atomique **7**

2 CUDA : *Compute Unified Device Architecture* **8**
2.1 Présentation et historique 8
2.2 Spécifications techniques 8
2.3 La gestion mémoire 10
2.3.1 La mémoire globale, et la coalescence 11
2.3.2 La mémoire partagée et les conflits de banques 13
2.3.3 La divergence des *warp* 13
2.4 Caractéristiques de CUDA 13
2.5 Exemple : addition de deux matrices 14

3 Implémentation d'un algorithme d'*Image Inpainting* **16**
3.1 Matériel utilisé 16
3.2 L'algorithme 17
3.2.1 Description fonctionnelle 17
3.2.2 Illustrations 17
3.2.3 L'implémentation C 17
3.3 Port vers l'architecture CUDA 19
3.3.1 Découpage de l'algorithme pour CUDA. 19
3.3.2 Premier *kernel* : Mise à 0 des valeurs d'un tableau 19
3.3.3 Implémentation CUDA des permutations pre *FFT* 20
3.3.4 Autres *kernels* : opérations point-à-point et *reorderings* . 23
3.3.5 La Transformée de Fourier Rapide 24
3.4 Optimisation des permutations : résultats et interprétations . . . 25
3.4.1 Les blocs de *threads* 25
3.4.2 L'impact de la coalescence 25
3.4.3 Les conflits de banques 26
3.4.4 Conclusion intermédiaire 27
3.5 La Transformée en Cosinus Discrète 27
3.6 Application de l'algorithme à différentes images 27
3.6.1 *Inpainting* sur le Fond diffus cosmologique 28
3.7 Performances de l'algorithme porté sur CUDA : théorie et pratique 28
3.7.1 Estimation des gains de l'*inpainting* avec CUDA 29
3.7.2 Gains finaux sur l'algorithme d'*inpainting* 30

4 Conclusion et Perspectives **32**
Glossaire 35
Bibliographie 36

A Codes **37**
A.1 Transposées 37
A.1.1 Transposée du SDK de CUDA 37
A.1.2 Adaptation à des donnï¿½es de type Complexes 37
A.2 Ensemble des kernels utiliés par l'algorithme d'*inpainting* avec CUDA 38
A.2.1 En-tête du fichier de *kernels* 38
A.2.2 Mise à 0 d'un tableau 39
A.2.3 Soustraction de 2 tableaux 39
A.2.4 Seuillage des coefficients d'un tableau 39
A.2.5 Opération comprenant une addition et une multiplication de tableaux 40
A.2.6 Preswap pré Transformée de Fourier Rapide 40
A.2.7 Recombinaison complexe post *FFT* 42
A.2.8 Recombinaison complexe pre *FFT* inverse 43
A.2.9 Permutations post *FFT* 45
A.2.10 *Kernels* utilitaires 46

B Illustrations **48**
B.1 Application de l'algorithme d'*inpainting* sur une image de la Sphère 48

Introduction

Le Calcul Haute Performance connaît un fort essor ces dernières années. Auparavant réservé aux simulations nucléaires ou climatiques, il s'étend maintenant à de nombreuses branches en répondant à de nombreux besoins différents. De la physique au monde de la finance en passant par les sciences du vivant, le HPC (Calcul Haute Performance) est devenu plus qu'un besoin, une nécessité.

Alors que les solutions étaient très complexes et coûteuses il y a peu, certaines se basent aujourd'hui sur des processeurs standards et des logiciels libres.

Dans cette course au meilleur rapport performances/prix, les GPU ont commencé à percer dès 2003, proposant des solutions de calculs peu coûteuses. Nvidia et ATI (maintenant AMD) ont chacun développé une API, respectivement CUDA et CMD, permettant d'utiliser au mieux la puissance de calcul de ces puces graphiques. En même temps, ils ont profité de l'augmentation quasi exponentielle de la puissance de leur puces. Néanmoins, les GPU n'ont pas les mêmes fonctions que les CPU. C'est pour cette raison que tout algorithme ne s'adapte pas forcément bien sur carte graphique, ces dernières exploitant leur parallélisme au maximum.

D'un autre côté, d'autres solutions sont actuellement étudiées. Le processeur d'IBM développé pour la PlayStation3, le CELL, est une vraie réussite. Il est logiquement entré dans le monde des superordinateurs, et de bien belle manière : il équipe le premier superordinateur dépassant le pétaflop. Les FPGA sont probablement la solution la plus intéressante en termes de performances car ils sont créés pour des utilisations bien particulières, mais le coût de développement et de production reste problématique. CUDA a l'avantage d'être utilisable simplement sans coût exhorbitant ni infrastructures particulières.

Le sujet de mon stage est motivé par le fait que CUDA doit bien s'adapter au traitement d'image. En effet, nombre d'algorithmes de ce domaine sont de type local, ce qui se prête parfaitement à la parallélisation. Le choix de CUDA n'est donc pas anodin.

Nous allons commencer par une présentation de CUDA. Ensuite, nous verrons au travers d'un algorithme très simple par quelles difficultés nous sommes passés pour l'implémenter sur le GPU. De manière générale, nous avons choisi cet algorithme afin de voir quels gains nous pourrions potentiellement espérer pour d'autres cas.

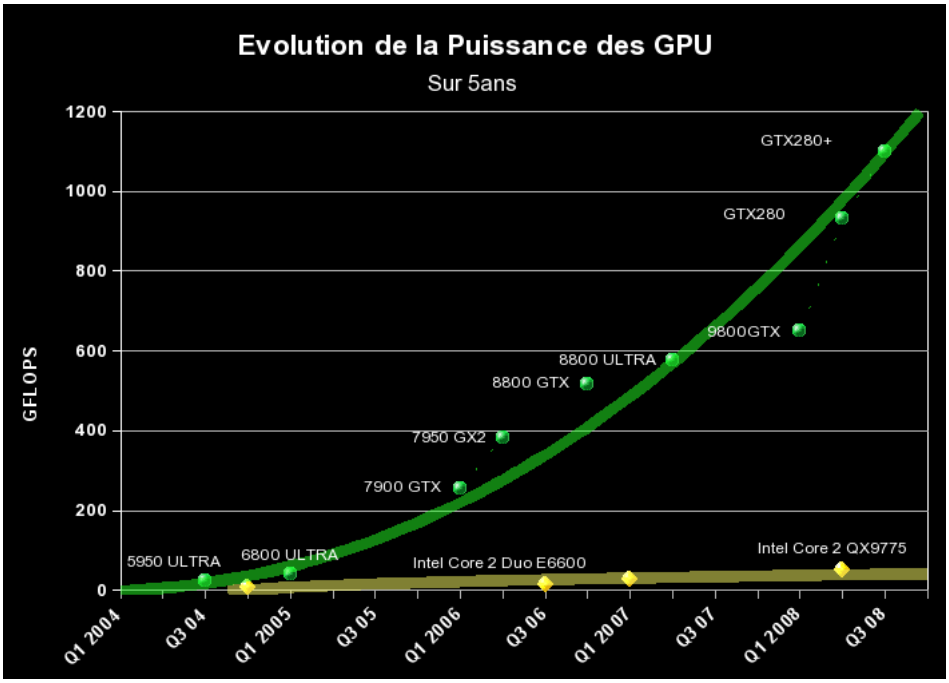


FIG. 1: On constate une évolution quasi exponentielle de la puissance brute de calcul des GPU sur ces 5 dernières années

Chapitre 1

Le Commissariat à l'Energie Atomique

Le **Commissariat à l'Energie Atomique** (CEA) est un organisme de recherche public français, né de la volonté du Général de Gaulle. Initialement destiné aux "recherches scientifiques et techniques en vue de l'utilisation de l'énergie nucléaire dans les domaines de la science, de l'industrie et de la défense nationale " !.

J'ai effectué mon stage sur le site de Saclay à l'**Institut de recherche sur les lois fondamentales de l'Univers** qui regroupe les départements d'astrophysique, de physique des particules, de physique nucléaire, et de l'instrumentation associée, appartenant à la **Direction des Sciences de la Matière** (DSM) du CEA.

Plus précisément j'ai été intégré dans **Service d'Electronique des Détecteurs et d'Informatique** (SEDI), dans le laboratoire du TRAPS.

TRigger Algorithme et Processeurs Spécialisés. Le coeur de métier de l'équipe c'est la conception de systèmes (architectures matérielles) d'aquisition de données dédiés Ã des expériences de physique. On utilise de plus en plus les FPGA ces dernières années principalement pour leur flexibilité, ils ont les avantages du software (programmabilité) et ceux du hardware (performances). Dans l'équipe les passerelles entre software et hardware sont nombreuses (on écrit souvent les software qui pilotent le hard développé) et s'intéresser à d'autres applications comme le calcul scientifique sur des architectures dites "spécialisées" est de ce point de vue naturel.

Chapitre 2

CUDA : *Compute Unified Device Architecture*

2.1 Présentation et historique

Avec l'arrivée des premières cartes graphiques à la fin des années 70, les idées ont commencé à germer. Les recherches se sont intensifiées dans le but d'utiliser ces cartes à des fins autres que l'affichage. Ces dernières années ont vu une considérable amélioration des performances, tant celles de calcul, que la bande passante du bus graphique qui utilisait auparavant un bus standard, puis qui s'est vu dédier le bus AGP, pour continuer aujourd'hui avec le bus PCI-Express x16. Le PCI-Express x16 est simplement 16 fois plus rapide que l'AGP x1 sorti en 1997. Ces évolutions incroyables ont donné des idées aux deux principaux acteurs du marché hautes performances, Nvidia et son concurrent ATI, qui ont décidé de proposer des moyens de programmer leurs puces de manières standard. Alors qu'avant cela, les recherches portaient sur l'utilisation des API de calcul existantes, Nvidia a décidé de proposer à ses clients une API développée spécifiquement pour le calcul haute performance. C'est de là que CUDA est né.

2.2 Spécifications techniques

CUDA n'est utilisable qu'à partir du G80 de Nvidia, soit la série des GeForce 8. Pour commencer, nous allons détailler l'architecture de ces *GPU*.

Le GeForce 8 est un ensemble de multiprocesseurs (SM pour *Streaming Multiprocessor*) indépendants équipés chacun de 8 processeurs généralistes (SP pour *Streaming Processor*) et de 2 processeurs spécialisés (SFU pour *Super Function Unit*). Les SP effectuent toujours une même opération à la manière d'une unité SIMD (*Single Instruction Multiple Data*). Un SM utilise les 2 types de processeurs pour exécuter des instructions sur des groupes de 32 *threads* (appelés des *warps*). Un *warp* est exécuté en 4 cycles d'horloge sur 8 ALUs. Le haut de gamme de la série possède 16 SM (voir la figure 2.1 pour un schéma détaillé).

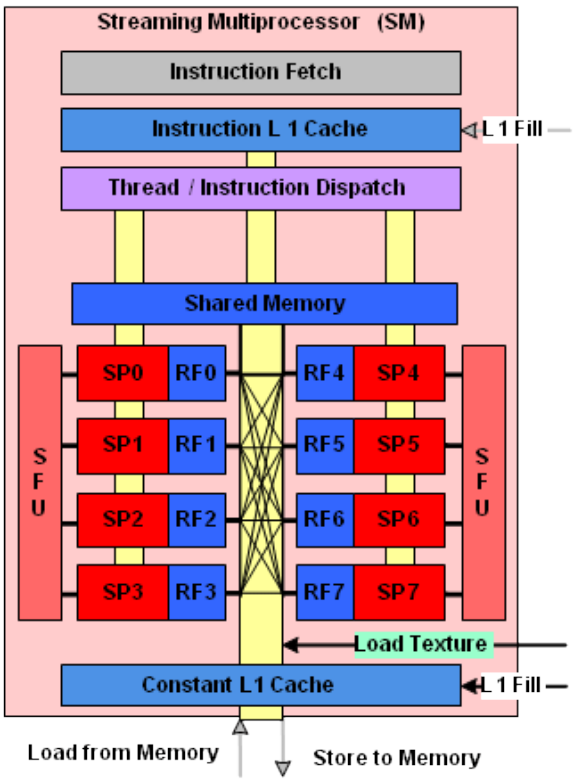


FIG. 2.1: Schéma d'un multiprocesseur.

.

Un *kernel* est un programme exécuté sur le GPU. C'est un ensemble de blocs composés de *warp* (maximum 16 *warps* par bloc, soit un maximum de 512 *threads* par bloc). Une mémoire partagée est disponible pour chaque bloc (16KB) ; seuls les *threads* d'un même bloc peuvent communiquer via celle-ci.

Résumé des limitations hardware du Geforce 8.

- *threads* par SM : 768
- *warps* par SM : 24
- blocs par SM : 8
- *threads* par bloc : 512
- registres 32 bits par SM : 8192
- mémoire partagée par SM : 16Ko
- constantes cachées par SM : 8Ko
- textures 1D cachées par SM : 8Ko

Afin de tirer les meilleures performances possible du GeForce 8, il faut organiser son code de manière à donner le plus grand nombre de *threads* possible tout en respectant les limites ci-dessus. Nvidia met à disposition divers outils qui permettent d'optimiser les choix.

L'*Occupancy Calculator* permet en fonction du nombre de *threads* par bloc et du nombre de registres utilisés par un *kernel*, d'avoir une estimation de la charge du GPU. Le nombre de *threads* est calculé simplement à l'aide de la taille du bloc, et le nombre de registres est donné, par une option du compilateur nvcc : *-cubin*. Cela génère un fichier avec le nombre de registres utilisés par un *kernel*, le taille de la mémoire partagée utilisée, etc...

Le *Visual Profiler* est l'outil d'analyse de performances. Comme GNU gprof (pour gcc), il donne des informations sur le temps passé dans chaque fonction. De plus il informe aussi de l'occupation du GPU par *kernel*, ainsi qu'accès à divers graphiques. Il donne aussi le nombre d'accès mémoire coalescents ainsi que des informations sur les branchements et la divergence (cf chapitre suivant) et quelques informations supplémentaires très utiles lors d'un développement.

2.3 La gestion mémoire

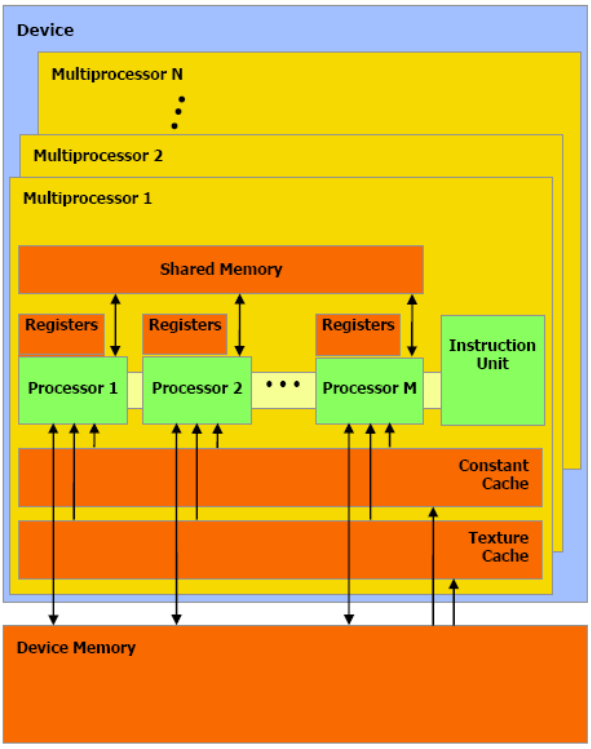


FIG. 2.2: Architecture mémoire des multiprocesseurs [12]

.

2.3.1 La mémoire globale, et la coalescence

Principes

Le GPU possède une mémoire globale partagée par l'ensemble des multiprocesseurs, à la manière de la RAM pour un CPU, ainsi que plusieurs mémoires caches. Chaque multiprocesseur possède un cache pour les constantes, un cache pour les textures, une mémoire partagée, ainsi que des registres. Un accès à la mémoire globale prend environ 400 cycles d'horloge, alors qu'un accès à la mémoire partagée est environ 100 fois plus rapide (environ la même vitesse qu'un accès à un registre).

La coalescence

Avoir des accès mémoire coalescents permet des gains de vitesse qui peuvent varier jusqu'à un facteur 20. La mémoire globale souffre d'une latence importante (400 à 600 cycles d'horloge pour un accès). Pour remédier à ce problème, CUDA donne la possibilité d'accéder à un bloc de plusieurs cases mémoires, une sorte de *burst*. On obtient des gains de vitesse si l'on accède à des cases mémoires voisines et dans l'ordre des indexes de *threads*. Les figures 2.3 et 2.4 illustrent ces propos.

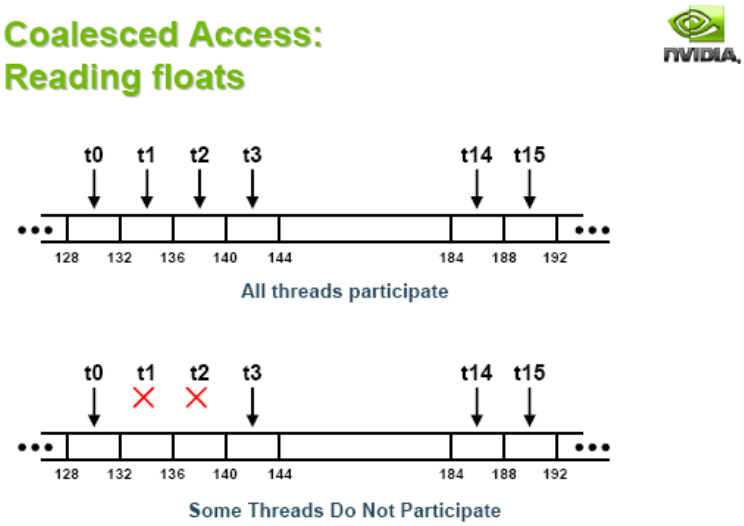


FIG. 2.3: Deux types d'accès coalescents [9]. La coalescence est effectuée au niveau d'un demi *warp* et ne peut être valide que si l'on accède à des régions mémoire de 64, 128 ou 256 octets (respectivement int-float, int2-float2, int4-float4). Il faut aussi que le *thread* 0 soit aligné sur une case mémoire multiple de la taille de la région. Le *thread* 1 accède donc à la 2ème case, le *thread* 2 à la 3ème, etc...

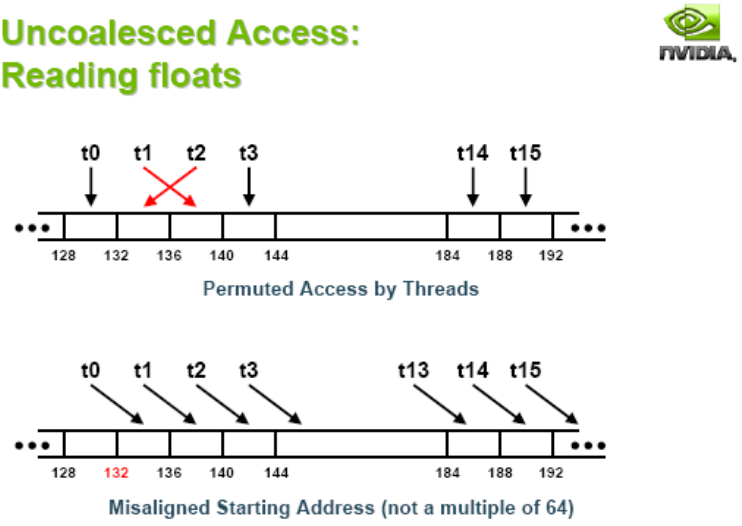


FIG. 2.4: Deux types d'accès non coalescents [9]. Le premier est dû au fait que les threads n'accèdent pas dans l'ordre à des cases voisines. Le second est dû à un problème d'alignement. Le premier thread d'un *warp* doit accéder à une case mémoire multiple de 64.

On peut voir sur ces schémas que la coalescence ne requiert pas uniquement que des accès à des cases voisines, mais aussi un alignement. Ce dernier doit être appliqué sur un multiple de 64 octets, et les régions mémoires doivent être

d'une taille de 64, 128, ou 256 octets. Il faut aussi prendre en compte que le G80 applique cette coalescence sur un demi *warp*, soit 16 *threads*, et donc 16 cases mémoire.

2.3.2 La mémoire partagée et les conflits de banques

Principes

La mémoire partagée est uniquement accessible par un bloc de *threads*. Les accès mémoires sont jusqu'à 100 fois plus rapide que dans la mémoire globale. La mémoire partagée devient utile lorsque l'on a plusieurs opérations à appliquer successivement dans un même bloc de données. La principale difficulté est qu'elle n'est pas partagée entre les blocs. Il faut donc que pour de meilleures performances que le bloc de *threads* exécute des opérations sur les données qu'il a rapatrié localement.

Les conflits de banques

Les accès à la mémoire partagée se font au travers de 16 banques mémoire. Pour être optimal, il faut que chacun des 16 *threads* d'un demi *warp* accède à une banque différente. Par exemple, si le *thread* 0 accède à la case 0 d'un bloc de mémoire partagée, et que le *thread* 1 accède à la case 16 et que ces deux cases sont accédées à partir de la même banque. On a donc un conflit de banque. L'exemple *transpose* du *SDK* montre une manière de les éviter (cf. transpose_kernel.cu).

2.3.3 La divergence des *warp*

On parle de divergence lorsque différents *threads* d'un même *warp* travaillent dans 2 branches différentes d'un même *if*. L'impact sur les performances peut être important car le GPU est obligé d'exécuter le *warp* sur plusieurs cycles d'horloge, alors que 2 cycles suffisent lorsque les *threads* d'un *warp* sont parfaitement SIMT.

2.4 Caractéristiques de CUDA

CUDA est une API de type haut niveau. Nvidia place son API comme une abstraction du hardware. Le but est de simplifier au maximum le travail du développeur. Il s'avère qu'une bonne compréhension du hardware est nécessaire à l'instar de son concurrent principal : ATI. De son côté, ce dernier proposait jusqu'à trés récemment, uniquement un accès à des fonctions bas niveau avec la CTM. Il propose maintenant lui aussi sa propre abstraction avec le *Brook+*, une modification propriétaire du langage *BrookGPU* [7].

Essayons de voir quels sont les points sur lesquels CUDA se différencie de ses concurrents. Voyons tout d'abord quelques points forts :

- CUDA utilise le langage C, avec quelques extensions.
- Permet des lectures/écritures à des adresses arbitraires (*gathering* / *scattering*) dans la mémoire globale.

– Une mémoire partagée est à disposition de chaque blocs de *threads* (16KB par bloc, non partagée entre les blocs). La bande passante est très élevée ce qui permet de l'utiliser à la manière d'un cache.
– CUDA propose une fonction d'allocation de mémoire contigüe sur l'hôte. Les copies, depuis, et vers l'hôte sont grandement accélérées de cette manière.
– Support complet des opérations sur les entiers et les opérateurs binaires.

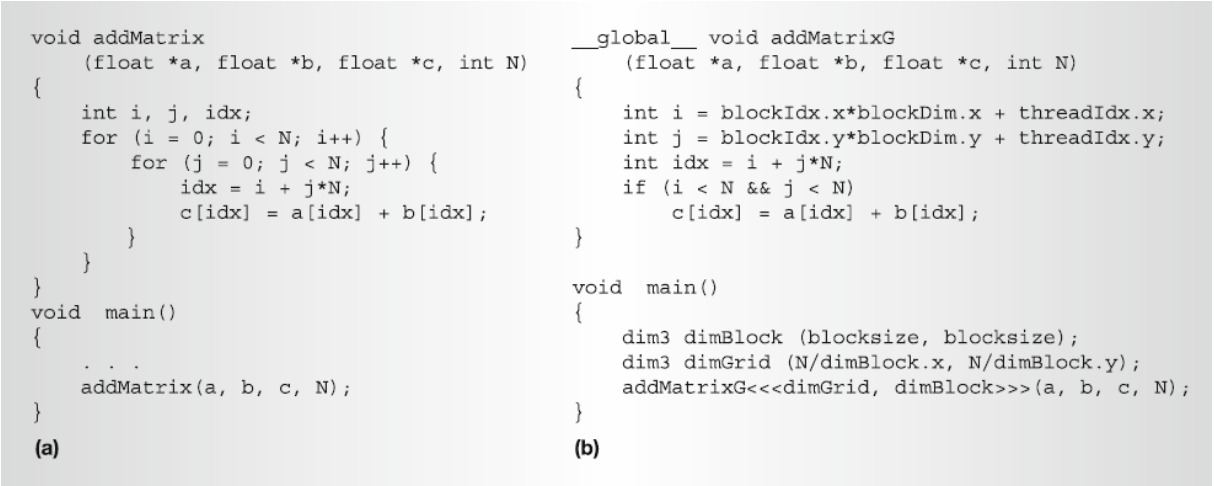
Toujours de manière non exhaustive, voici quelques problèmes qui se posent :
– Les fonctions récursives ne sont pas supportées.
– Etant données les limitations du bus PCI-Express, il est préférable de privilégier les calculs intensifs par rapport aux accès mémoire. Cela ne s'adapte donc pas à tous les types d'algorithmes.
– Les *threads* fonctionnent par groupes de 32 (un *warp*) et ont besoin d'exécuter des instructions similaires, simultanément. Ce fonctionnement entraîne une grosse perte de performance pour des tâches divergentes.
– Nvidia a fait le choix de ne pas suivre totalement les normes de l'IEEE 754, probablement pour de complexité de mise en oeuvre et par conséquent de place dans la puce. Les opérations concernées sont la division et la racine carrée qui sont d'une précision légèrement moindre à la simple précision.

Il faut savoir que Nvidia devrait corriger ces problèmes de normes dans ses prochaines générations de GPU. Enfin, concernant le PCI-Express, la version 2.0 double le débit théorique du bus, et la version 3.0 est déjà prévue pour 2010 avec à nouveau un doublement de la bande passante. Cela reste tout de même problématique pour certains programmes où il est nécessaire de faire de nombreux allé-retours entre le CPU et le GPU, ainsi que pour la programmation multi-GPU qui, suivant les algorithmes, demande souvent des échanges de données entre les GPU. Le problème devrait donc persister. Il est donc nécessaire dans un programme CUDA d'effectuer suffisamment de calcul afin que les transferts mémoire soient insignifiants.

2.5 Exemple : addition de deux matrices

La figure 2.5 présente une comparaison entre un code C standard, et un code C CUDA pour l'addition de deux tableaux. On peut voir que le code CUDA n'est pas particulièrement complexe. Chaque *thread*, en fonction de son indice dans son bloc, calcule l'indexe auquel il va travailler dans le tableau. Dans le cas d'une opération point-à-point comme l'addition, il suffit de rapporter l'indice du *thread* à son incide correspondant dans la matrice, en fonction de l'indice de son bloc.

L'architecture des GPU Nvidia leur est propre et les optimisations que l'on peut apporter sont directement liées à cette architecture. CUDA demande de savoir gérer plusieurs paramètres pour espérer optimiser correctement son code. La coalescence, la taille de la grille et des blocs et par conséquent le nombre de *threads*, de registres et la quantité de mémoire par blocs, ainsi que les conflits de banques, sont les principales sources d'optimisations.



```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

FIG. 2.5: Implémentation C standard (a) et CUDA (b) d'un programme qui additionne deux tableaux [10]. La fonction *addMatrixG* est appelée avec des paramètres supplémentaires relatifs à CUDA. $(N/blockDim.x * N/blockDim.y)$ blocs sont appelés, avec chacun $blocksize * blocksize$ *threads*. On lance un *thread* par élément à calculer, soit $N * N$ threads dans notre cas.

Ecrire un code CUDA n'est pas plus long que d'écrire un code C. En revanche, adapter les algorithmes déjà existants à l'architecture particulière des GPU afin d'obtenir des performances intéressantes n'est pas évident, et demande un important temps de développement.

Développer pour l'achitecture CUDA n'est pas chose évidente. Même lorsqu'un algorithme se parallélise simplement, les spécificités hardware du GPU font qu'il est parfois nécessaire de passer par plusieurs étapes d'optimisations qui ne sont pas toujours aisées. En contre partie, il est souvent possible d'adapter une optimisation faite sur un code pour un autre. Ceci est dû au fait que les optimisations sont souvent directement liées au *hardware*.

Chapitre 3

Implémentation d'un algorithme d'*Image Inpainting*

Dans le cadre de mon stage, il m'a été proposé de porter un algorithme de traitement d'image, sur GPU à l'aide de CUDA. Cet algorithme s'avère être un bon exemple car il utilise certaines briques de bases de ce domaine. Son intérêt principal est sa simplicité d'implémentation, ce qui permet une première approche de CUDA plus aisée.

L'*Image Inpainting* est une méthode itérative permettant de réparer des images abimées ou de retirer des éléments non désirés sur des images.

Nous allons commencer par présenter le matériel et les outils utilisés, puis nous verrons de quelle manière l'algorithme s'implémente. Enfin, nous verrons ensuite diverses optimisations appliquées, et enfin nous terminerons par l'interprétation des résultats.

3.1 Matériel utilisé

Nous avons effectué l'ensemble de notre développement sur 2 machines différentes :

- Un portable équipé d'une carte graphique Quadro FX370 (seulement 16 *Stream Processor*).
- Un desktop équipé d'un Core2Duo 2,66GHz et d'une Quadro FX4600 (768mo de RAM et 112 *Stream Processor*).

Dans le cadre de certains tests, nous avons pu comparer avec une carte légérement plus puissante sur la machine de Frederic Chateau, équipée d'une 8800 GTX (768mo RAM, 128 SP).

Nous avons utilisé un serveur Subversion local qui nous a bien été utile lors des changements de machines, et plus particulièrement lors de mauvaises manoeuvres ou nous avons eu à utilisé la fonction *revert* à plusieurs reprises.

3.2 L'algorithme

3.2.1 Description fonctionnelle

Prenons par exemple une carte du ciel en astrophysique. Si nous souhaitons masquer les sources très lumineuses, on peut simplement enlever la couleur à ces zones (mettre une valeur fixe pour ces pixels), ou alors, une alternative que propose l'algorithme d'*inpainting* est de remplir cette zone à l'aide de son voisinage (plus ou moins grand) afin d'avoir un remplissage homogène à ce voisinage. C'est en quelque sorte la texture que l'on étendrait à l'intérieur du masque sans pour autant modifier les zones hors masque.

3.2.2 Illustrations

L'*inpainting* peut être utilisé pour tout types d'images. La figure 3.1 donne un exemple sur *Barbara* alors que la figure 3.9 illustre l'application de l'algorithme sur le Fond Diffus Cosmologique.

FIG. 3.1: Un *inpainting* est appliqué sur une image à laquelle il manque 50% de ses pixels.

3.2.3 L'implémentation C

Comme on peut le voir sur l'algorithme 1, à chaque itération, on effectue une transformée en cosinus discrète (DCT) sur l'ensemble de l'image, suivi d'un seuillage des coefficients et on termine par une DCT inverse. Le nombre d'itérations ainsi que les valeurs de seuils sont des paramètres à affiner suivant

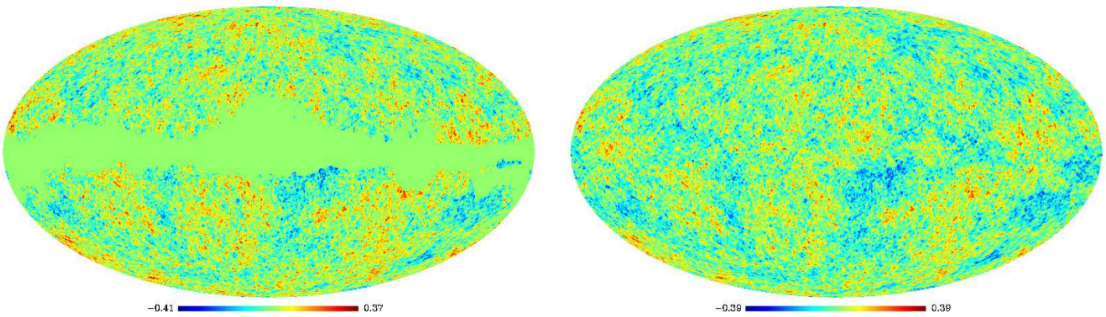


FIG. 3.2: **A gauche :** Carte provenant du satellite WMAP. Les zones sur le plan galactique et les points d'où proviennent de fortes sources radios ont été masqués. **A droite :** Carte obtenue en appliquant un algorithme d'*inpainting* sur la sphère.[6]

Algorithm 1 *Image Inpainting* pour CPU

Require: $K, n \in \mathbb{N}$, $\mathbf{image\_in}_{n,n}(\mathbb{R})$, $\mathbf{image\_out}_{n,n}(\mathbb{R})$
Require: $K, n \in \mathbb{N}$, $\mathbf{residual}_{n,n}(\mathbb{R})$, $\mathbf{alpha}_{n,n}(\mathbb{R})$, $\mathbf{mask}_{n,n}(\mathbb{R})$
Require: $seuil\_min, seuil\_max \in \mathbb{N}$
  $delta\_seuil = seuil\_max - seuil\_min)/K$
  for $i = 0$ to K **do**
    $\mathbf{residual} = \mathbf{image\_in} - \mathbf{image\_out}$
    $\mathbf{alpha} = DCT(\mathbf{image\_out} + \mathbf{mask} * \mathbf{residual})$
    $\mathbf{alpha} = \mathbf{alpha} * (abs(\mathbf{alpha}) >= seuil)$
    $\mathbf{image\_out} = invDCT(\mathbf{alpha})$
    $seuil = max(0, seuil - delta\_seuil)$
  end for
  return *image_out*

l'image. Le masque indique les zones de données manquantes sur lesquelle on souhaite appliquer l'algorithme.

Il est tout à fait possible d'utiliser un autre opérateur que la DCT. On peut appliquer ce type d'algorithme avec des FFT ou encore des transformées en ondelettes.

Pour commencer, j'ai implémenté une version en C de l'algorithme à l'aide de la librairie FFTW3. Cette librairie inclue directement les opérateurs de DCT. CUDA, de son côté ne possède pas de librairie aussi complète que FFTW3 et n'inclue pas de DCT. En revanche CUDA possède CUFFT, une libraire qui implémente des transformées de Fourier.

Objectif de mon travail : Coder une DCT à partir de la libraire CUFFT.

La DCT est un opérateur très proche de la FFT. Afin d'obtenir une DCT à partir d'une FFT, il suffit d'ajouter une étape de permutation avant la FFT, ainsi qu'une recombinaison à l'aide des parties complexes après la FFT. On applique le même raisonnement inversé pour la DCT inverse. Il existe différentes

manières de faire ces permutations. L'avantage de celle choisie est qu'elle ne nécessite pas d'espace supplémentaire (certains algorithmes requièrent de doubler l'espace mémoire). Elles sont décrites par John Makhoul dans son article [11]. C'est d'ailleurs l'algorithme utilisé par la librairie FFTW3 (cf. *reodft010e-r2hc.c*).

3.3 Port vers l'architecture CUDA

3.3.1 Découpage de l'algorithme pour CUDA.

Etant donné le peu d'opérations comprises dans la boucle de l'algorithme 1, nous avons décidé de porter l'ensemble de la boucle sur le GPU. Ce choix donne l'avantage d'éviter les allés retour sur le CPU et permet donc de profiter pleinement du GPU. En revanche, nous avons dû effectué un découpage différent de l'algorithme 1. L'obligation de passer par des *kernel*s pour travailler sur les données présentes sur la mémoire du GPU nous a donc demandé quelques modifications. L'algorithme 2 montre le découpage effectué.

Algorithm 2 *Image Inpainting* pour GPU

Require: $K, n \in \mathbb{N}$, $\mathbf{image\_in}_{n,n}(\mathbb{R})$, $\mathbf{mask}_{n,n}(\mathbb{R})$
Require: $seuil\_min, seuil\_max \in \mathbb{N}$

$delta\_seuil = seuil\_max - seuil\_min)/K$
$init\_to\_zero\_kernel(\mathbf{image\_out})$

for $i = 0$ to K **do**
  $minus\_kernel()$
  $init\_DCT\_kernel()$

  $preswap\_kernel()$
  $FFT()$
  $complexSwap\_kernel()$

  $abs\_kernel()$

  $complexSwap\_inverse\_kernel()$
  $inverse\_FFT()$
  $postswap\_kernel()$

  $seuil = max(0, seuil - delta\_seuil)$
end for
return $image\_out$

3.3.2 Premier *kernel* : Mise à 0 des valeurs d'un tableau

J'ai débuté par un simple portage du code sur CUDA, en supprimant les boucles, sans aucune autre réflexion quant aux performances que cette implémentation pourrait donner. J'ai donc créé différents *kernel*s pour les opérations sur les tableaux. Pour une opération point-à-point, CUDA permet de parallèliser le code sans aucun effort. L'API permet une association directe des blocs de

threads par rapport à des blocs de données de la matrice. Par conséquent, cela permet de dérouler des boucles parcourant un tableau de données aisément. Initialiser un tableau à une valeur donnée se fait directement en associant chaque *thread* à une valeur de ce tableau.

Comme je l'ai expliqué précédemment, on donne une opération à réaliser à chaque *thread* en fonction de son indice. Considérons cet exemple :

```
__global__
void zeros_kernel(Complex * data, int width)
{
  unsigned int xIndex =
        blockIdx.x * BLOCK_DIM + threadIdx.x;
  unsigned int yIndex =
        blockIdx.y * BLOCK_DIM + threadIdx.y;

  unsigned int index = xIndex + yIndex * width;

  data[index].x = 0;
  data[index].y = 0;
}
```

On veut initialiser à 0 un tableau de n complexes. Pour fonctionner à charge maximum le GPU nécessite un grand nombre de *threads* (plusieurs milliers). Dans notre cas, on a des tableau d'une taille minimum de $128^2 = 1,6.10^4$ éléments. On fait donc en sorte qu'un *thread* s'occupe d'un complexe.

Ce *kernel* est intéressant dans la mesure ou il existe une fonction de l'API qui à le même effet tout en étant moins rapide : *cudaMemset()* [5].

La coalescence est opérationnelle pour ce *kernel*. C'est vrai car un Complex est de type float[2]. Normalement, les instructions dans un demi *warp* sont synchronisées. Il devrait donc d'abord remplir les champs x à l'adresse index (associée au *thread*), puis ensuite les champs y. Dans ce cas, 2 *threads* voisins n'accèderaient pas à deux cases mémoire voisines, mais à une case sur 2 (en considérant qu'on sépare les 2 champs de la structure). Ce ne serait donc pas coalescent. Mais nvcc optimise les cas de vecteurs à 2 et 4 valeurs (int2, float2, int4, float4, etc...).

Comme nous verrons après, si jamais on souhaite ne remplir qu'un champ sur les 2, il faut faire une opération, même sans effet sur le 2ème champ afin d'assurer la coalescence. Par exemple, si on enlève la dernière ligne du code ci dessus, les écritures ne sont plus coalescentes.

3.3.3 Implémentation CUDA des permutations pre *FFT*

Nous avons choisi d'optimiser le premier *kernel* de permutations. En considérant un tableau 1D, il envoie les valeurs situées en indice pair dans la première

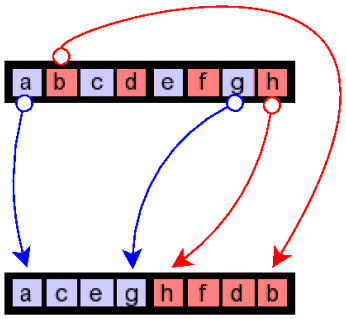


FIG. 3.3: Permutations executées avant la *FFT* pour le calcul de la *DCT*. Les indices pairs sont basculés dans l'ordre croissant en début de tableau. Les indices impairs le sont aussi, mais dans l'ordre décroissant et en fin de tableau.

partie du tableau, et les valeurs d'indices impairs, dans la seconde. Les figures 3.3 et 3.4 illustrent ces permutations, d'abord en 1D puis en 2D.

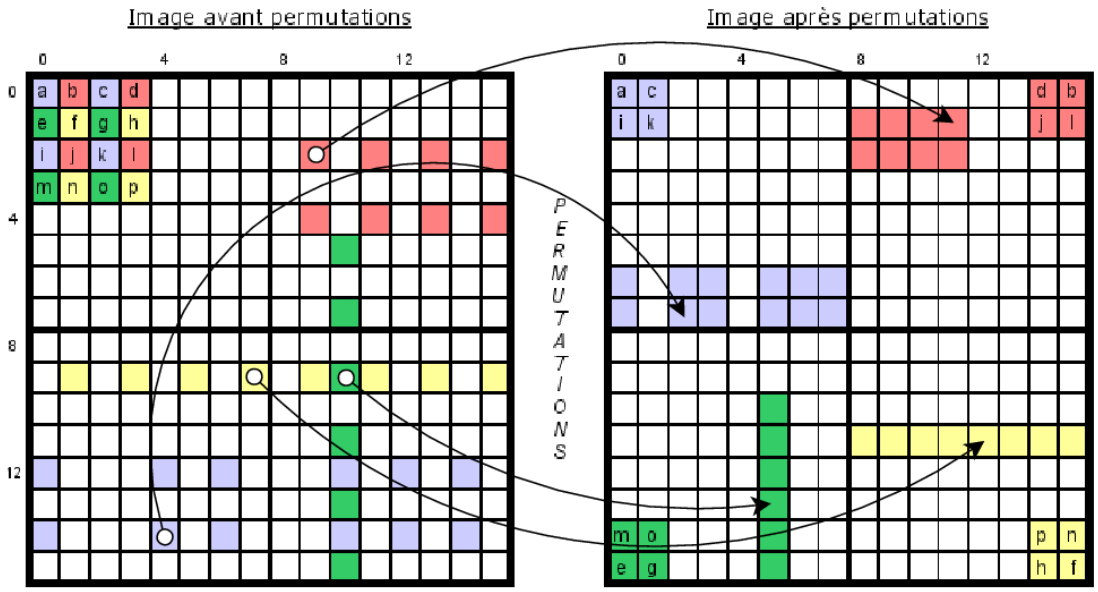


FIG. 3.4: Généralisation 2D des permutations.

Réorganisation de l'image : Suivant la position du pixel de l'image initiale, il est permuté différemment :

1. x pair, y pair : partie haut gauche de la sortie, dans l'ordre croissant selon x et y. (en bleu sur l'image)
2. x pair, y impair : partie haut droite de la sortie, dans l'ordre croissant selon y, décroissant selon x (en rouge).
3. x impair, y pair : partie bas gauche de la sortie, dans l'ordre croissant selon x, décroissant selon y (en vert).

4. x impair, y impair : partie bas droite de la sortie, dans l'ordre décroissant selon x et y (en jaune).

threadIdx.x	threadIdx.y	Bloc Index	Global IN	Global OUT
0	0	0	0	0
1	0	2	2	1
2	0	4	4	2
3	0	6	6	3
4	0	7	7	12
5	0	5	5	13
6	0	3	3	14
7	0	1	1	15

FIG. 3.5: Exemple donnant la permutation des indexes avec une image 16x16 et une taille de bloc 8x8. On considère le bloc d'indice (0,0) sur la grille et ce, uniquement pour les lignes paires.

Si on lit les valeurs de la matrice successivement, on va devoir écrire à des emplacements successivement différents, ce qui va entraîner des écritures non coalescentes. On se propose donc d'utiliser la mémoire partagée pour effecuter la transition et réorganiser les données de manière à écrire des blocs de valeurs d'indexes voisins.

La figure 3.5 illustre comment les *threads* sont gérés pour les lignes paires de la matrice d'entrée. Les lignes paires remplissent "le haut" de la matrice, tandis que les lignes impaires remplissent "le bas". En fonction de l'indexe du *thread*, et du bloc où on se trouve, on calcule les indices d'accès au bloc de mémoire partagée et les indices d'entrée et de sortie de la matrice.

Pour commencer, on fait une copie en mémoire partagée d'un bloc de la matrice initiale. Ce bloc correspond au bloc de *threads* que l'on a choisi (64x2, nous verrons après comment nous avons fait ce choix). Pour une ligne paire, la première moitié des *threads* va travailler sur les pixels pairs, et les suivants sur les pixels impairs. On applique le même raisonnement pour les lignes impaires. La posititon du thread en y (variable *threadIdx.y*) permet de séparer les 2. On a une succession de cas selon que l'indice de ligne est paire ou non, et si l'indice du pixel est pair ou non.

Voyons maintenant comment on choisit les dimensions du bloc de *threads*. On a besoin de 16 *threads* pour effectuer des écritures coalescentes, donc on ne considère pas les cas ou la dimension en X du bloc est inférieure à 16 :

- **BLOCK_DIM_X** = 16 : pour chaque ligne, on va séparer en deux parties les écritures, soit 2x8 écritures globales (non coalescent). On a aussi un problème car les *warps* vont être divergents.

- **BLOCK_DIM_X** = 32 : on a bien ici deux écritures de 16, mais on a toujours le problème de divergence des *warps*.
- **BLOCK_DIM_X** = 64 : les écritures sont coalescentes, et un même *warp* est bien dans la même branche.
- **BLOCK_DIM_X** = 128 : Même chose que pour 64.

On a pris BLOCK_DIM_X = 64 car il fallait une suffisemment grande longueur pour que 16 *threads* puissent écrire de manière coalescente.

Concernant BLOCK_DIM_Y, on l'adapte de manière à obtenir la meilleure occupation GPU possible. D'après l'*occupancy calculator* et le *profiler*, pour 128 ou 256 *threads* par bloc, l'occupation était de 100%. Trois choix se proposent : 64x2, 64x4 et 128x2. Dans la pratique, il semble que 64x2 soit plus rapide, mais seulement d'un ordre inférieur à 1%.

Nous allons voir dans la partie suivante quelle influence peut avoir la taille des blocs. Avant cela, voyons les autres optimisations qui ont été faites.

3.3.4 Autres *kernels* : opérations point-à-point et *reorderings*

La permutation que l'on vient de voir est aussi effectuée à l'inverse après *FFT* inverse. L'optimisation est très proche de la précédente. On utilise la mémoire partagée pour réorganiser les données et pouvoir les écrire de manière coalescente. Le résultat de cette optimisation est même meilleur que pour la première permutation car on utilise seulement 9 registres par thread, au lieu de 11, ce qui permet une occupation totale du GPU.

Concernant les kernels exécutant des opérations linéaires point-à-point, ils sont déjà optimisés de base. C'est dans ce cas que CUDA est très pratique. On applique naïvement la grille de blocs au tableau en associant un *thread* par case mémoire. Seul d'un des *kernel*s n'est pas optimisé. On veut créer un tableau de réels à partir des parties imaginaires d'un tableau de complexes. Considérons le code suivant :

```
__global__
void complex2real_kernel(Complex* in, Real* out,
                         int width, int height)
{
  int xIndex = __umul24(blockIdx.x, blockDim.x)
                                        + threadIdx.x;
  int index  = __umul24(blockIdx.y, width) + xIndex;

  volatile float2 a;

  a.x = in[index].x;
  a.y = in[index].y;

  out[index] = a.x;
```

```
}
```

Le mot clé *volatile* sert à forcer le compilateur à utiliser un registre pour la variable *a*. Sans cela, il optimise en considérant que cette variable est inutile, et force donc des accès en lecture sur *in* qui sont non coalescents.

Les *reordering* complexes : Ils utilisent les propriétés de symétrie hermitienne de la transformée de Fourier. Le problème avec cette symétrie est qu'elle n'est pas alignée en terme de mémoire. Afin de gérer les problèmes d'alignements, il est nécessaire d'introduire des modulos. Après plusieurs tentatives de passage en mémoire partagée, pour éviter les accès non coalescents, il s'avère que la version naïve est systématiquement plus rapide qu'une version sans conflits mais qui demande plus de ressources pour gérer le non alignement.

3.3.5 La Transformée de Fourier Rapide

Tous les tests ont été effectués avec CUFFT, qui comme nous l'avons dit plus haut est déjà optimisée. Nous avons trouvé sur le forum CUDA de Nvidia, un code de *FFT* pour un signal 1D de 512 complexes avec un gain annoncé supérieur à 3 par rapport à CUFFT [13]. Cette implémentation a la particularité de lancer plusieurs *FFT* en même temps (*batched*). Dans notre cas, ca ne nous posait pas de problèmes. Nous avons simplement implémenté une version 2D à l'aide cette *FFT*.

Pour rappel, une *FFT* 2D correspond simplement à appliquer une *FFT* 1D sur chaque ligne, puis sur chaque colonne. On applique donc une transposée après avoir appliqué la *FFT* sur les lignes, puis on retranspose après une 2ème série de *FFT*. La *FFT* 2D inverse se calcule de cette façon :

$$conj(FFT(conj(X)))$$

Nous avons utilisé la transposée proposée par le SDK de CUDA. Nous l'avons modifiée, car elle était conÇue pour la transposition d'une matrice de flottants, alors que nous avions besoin de transposer des complexes. Pour se faire, nous avons ajouté un 2ème tableau en mémoire partagée, le premier utilisé pour la partie réelle, le second pour la partie imaginaire (les codes sont disponibles en annexe).

En 2D on obtient un gain de facteur 2 par rapport à la *FFT* 512 de CUFFT. Cela montre que la librairie CUFFT n'est pas optimale. Il faut savoir que les optimisations de V. Volkov ne sont valables que pour des *FFT* de longueur 8, 64 et 512. Adapter le code pour d'autres tailles de signaux serait probablement un travail très long car les choix qu'a fait V. Volkov pour optimiser sa libraire sont au plus proche des limites du hardware (utilisation des registres, occupation du GPU, coalescence, etc...).

Au total, notre implémentation de *FFT* 2D 512 sur GPU, est 8 fois plus rapide que la même transformation exécutée sur CPU avec la librairie FFTW3.

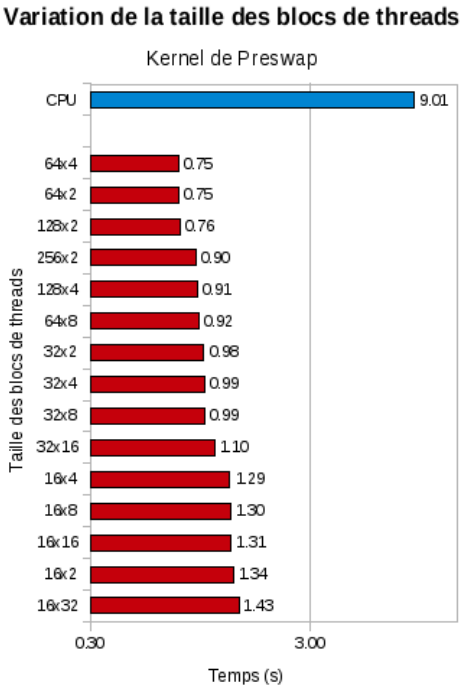


FIG. 3.6: Temps moyen pour l'application du Preswap pré *FFT* sur une image 1024x1024 en faisant varier les dimensions du bloc. Les valeurs en ordonnées sont les dimensions du blocs de *threads*. D'après l'*Occupancy Calculator*, c'est pour 128 ou 256 *threads* par bloc que l'on obtient la meilleure occupation du GPU. Cela se constate très bien dans les résultats ci-dessus. Ce kernel est directement lié à la bande passante mémoire, car il n'effectue que très peu de calcul. Dans ce cas, plus haut le taux d'occupation est, plus rapide sera l'exécution. *HOST* représente le temps d'exécution sur CPU et n'est présent sur ce graphique que pour comparaison.

3.4 Optimisation des permutations : résultats et interprétations

3.4.1 Les blocs de *threads*

Comme expliqué précédemment, la raison principale qui explique les différences que l'on peut voir sur la figure 3.6 est le taux d'occupation du GPU. Ce taux est calculé en fonction du nombre de registres utilisés et de la mémoire partagée utilisée et bien sur du nombre de *threads* par bloc. Le nombre de registres par *threads* ainsi que la mémoire partagée utilisée peuvent être récupérés avec l'option *-cubin* de nvcc.

Cette implémentation utilise 11 registres par thread. Avec l'utilisation de mémoire partagée, le GPU est utilisé à 83%. Si on pouvait réduire à 10 registres par thread, on obtiendrait un taux d'occupation du GPU de 100%. Ces valeurs proviennent de *Occupancy Calculator* et sont calculées en fonction du nombre de threads par blocs, du nombres de blocs par multiprocesseurs, etc...

3.4.2 L'impact de la coalescence

Pour commencer, nous avons simplement copié le code C dans CUDA. Notre première implémentation ne prenait pas compte de la coalescence, l'impact sur les performances s'en est directement ressenti. Nous avons donc ensuite indexé mes *threads* de manière à parcourir la mémoire en ligne et non en colonnes, et là, nous avons pu constater un gain de performances énormes.

La figure 3.7 compare les performances entre les versions suivantes :
- Host : version CPU.
- Naïve non coalescente : Aucun accès mémoire n'est coalescent.
- Naïve coalescente : seuls les accès en lecture sont coalescents.
- Optimisée : Tous les accès mémoire sont coalescents. Seuls quelques restent quelques conflits en mémoires partagée.

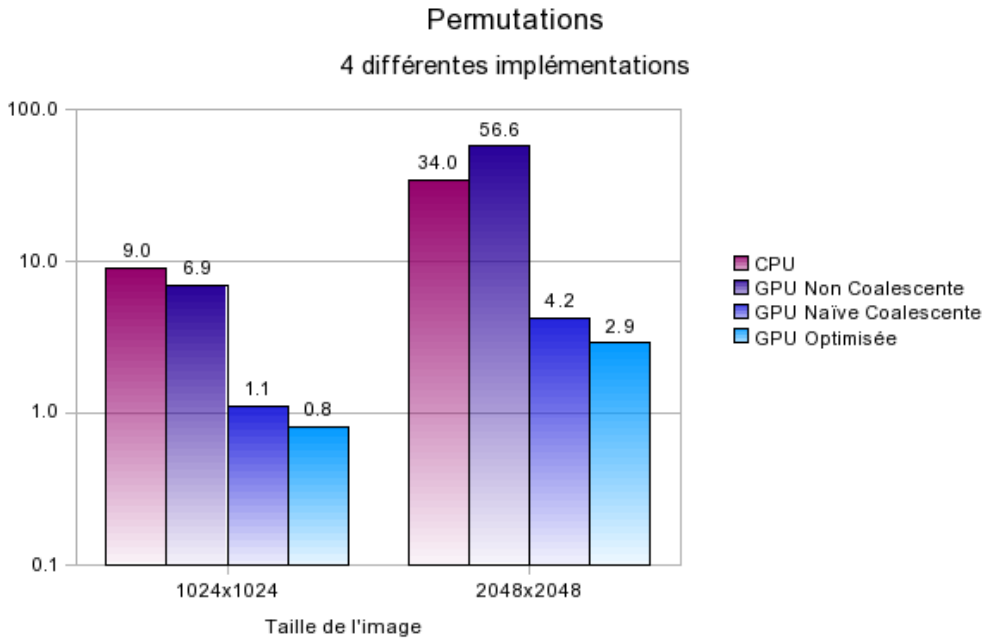


FIG. 3.7: Temps de calcul des permutations pre *FFT* selon 4 différentes implémentations. Toutes les valeurs des pixels de l'image sont permutées entre elles de la manière vue sur la figure 3.4. On voit immédiatement que les accès non coalescents sont très fortement pénalisants et par conséquent que certaines implémentations GPU peuvent donner des résultats très mauvais. Les temps de transferts mémoire ne sont pas inclus car dans l'implémentation globale on ne fait pas plusieurs aller-retours.

On peut même voir sur la figure 3.7 que la version CPU est plus rapide à s'exécuter que la version GPU non coalescente pour une image de 2048x2048. L'optimisation n'apporte qu'un gain assez faible, ce pour plusieurs raisons :
- l'ensemble des écritures de la version naïve étaient déjà coalescentes.
- le kernel est complètement lié à la bande passante mémoire. Les seuls calculs effectés sont les calculs d'indices.

3.4.3 Les conflits de banques

Une optimisation supplémentaire possible sur ce kernel porte sur la mémoire partagée. En effet, l'indice d'accès à la mémoire partagée avec un pas de 2. Comme on a dit précédemment, le GPU dispose de 16 banques mémoire. Le

thread 0 va accéder à la même banque mémoire que le thread 8 (1 et 9, 2 et 10, etc...). Il y a plusieurs façon d'éviter ces conflits, et l'une de ces manière est d'utiliser deux tableaux séparés en mémoire partagée.

3.4.4 Conclusion intermédiaire

L'implémentation de ce kernel est quasiement aboutie. Supprimer totalement les conflits de banques n'apportera que peu de gains (qu'on estime de l'ordre de 10%). On est totalement limité par la bande passante mémoire.

On constate aussi que parfois, optimiser n'est pas forcément évident. Vouloir à tout prix supprimer les accès non coalescents n'est pas forcément synonyme de gain de performances. La simplicité des permutations n'est pas reflétée quant au travail qu'il a fallu effectuer pour les optimiser. Il y a un réel besoin de comprendre ce que fait le code pour le porter à l'aide de CUDA.

3.5 La Transformée en Cosinus Discrète

Pour implémenter la Transformée en Cosinus Discrète, nous nous sommes basés sur notre implémentation sur CPU de l'algorithme de Makhoul [11] codée en C. Sur CPU, nous avons utilisé la librairie FFTW3 afin de réaliser les FFT. Cette implémentation de la DCT est entre 3 et 7 fois plus lente que la DCT de la librairie FFTW3 pourtant basée sur le même algorithme. Ceci est bien sûr dû aux nombreuses optimisations que cette librairie inclut. Le facteur 3 provient de tests réalisés sur des images de 128x128 alors que le facteur 7 provient d'une image de 2048x2048.

Il est important de prendre en compte ce que l'on vient de dire. Bien sûr, il est pertinant de vouloir comparer notre résultats par rapport à ce qui se fait de mieux sur CPU, mais il est tout aussi important de comparer par rapport à une implémentation donnée. Dans notre cas, entre notre implémentation CPU et son implémentation GPU, on pourra constater des accélérations entre 3 et 7 fois supérieurs.

3.6 Application de l'algorithme à différentes images

Nous avons commencé par appliquer notre algorithme à deux images, l'une bien connue dans le traitement d'image, l'autre moins, mais qui est très pertinante. La figure 3.8 montre ces exécutions. On peut voir que dans le cas de l'*inpainting* sur *barbara*, l'image n'est pas correctement recomposée. Nous n'avons pas pris le temps d'ajuster correctement les paramètres du seuillage. De plus la DCT n'a pas forcément les propriétés adéquates pour recomposer ce type d'images. Le deuxième exemple montre un *inpainting* sur une image horizontalement et verticalement très régulière. Dans ce cas, nous avons pu constater que nous ne voyons plus les différences entre l'image d'entrée et celle de sortie.

FIG. 3.8: *Inpainting* sur deux images différentes. L'une sans motifs réguliers de manière globale, l'autre avec. On constate que la *DCT* reconstruit très nettement l'image régulière tandis qu'elle ne reconsitue pas aussi bien *barbara*.

3.6.1 *Inpainting* sur le Fond diffus cosmologique

Actuellement, les astrophysiciens du Service d'Astrophysique de Saclay (SAP) utilisent ce type d'algorithme. Ils l'appliquent sur le fond diffu cosomologique pour masquer certaines sources d'energie. Nous avons donc utilisé leur données pour appliquer notre algorithme. Nous avons décomposé en sous images une image de la sphère à l'aide de la librairie *cfitsio Healpix*. La figure 3.9 représente l'application de l'algorithme implémenté avec CUDA, sur cette dernière.

3.7 Performances de l'algorithme porté sur CUDA : théorie et pratique

Nous avons mesuré les performances à l'aide des outils de mesure disponibles dans l'API CUDA. Afin de vérifier leur exactitude, nous avons effectués certains tests de manière aléatoire avec les librairies standard de mesure de temps. Tous

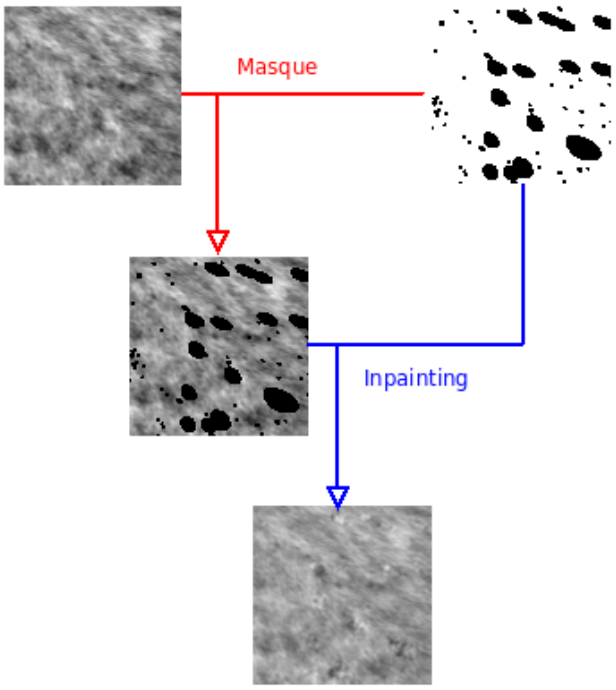


FIG. 3.9: *Inpainting* sur le Fond Diffus Cosmologique. Etape 1 :On applique le masque voulu sur l'image d'entrée. Etape 2 :On exécute l'algorithme d'*inpainting* sur l'image masquée, uniquement sur les zones correspondant au masque.

nos tests se sont recoupés.

3.7.1 Estimation des gains de l'*inpainting* avec CUDA

L'une des premières étapes du projet a été d'estimer les gains potentiels que nous pourrions avoir. Nous avions un ordre d'idée des gains que la librairie CUFFT pouvait nous apporter qui étaient compris entre un facteur 2 et 6 suivant les dimensions [2].

Nous avons effecuté quelques tests pour estimer quel facteur nous pourrions obtenir sur les opérations point-à-point (hors DCT) et nous avons constaté, qu'étant donné le peu de calcul qu'elles opéraient, on obtenait un facteur compris entre 20 et 30. S'ajoutent à cela les *reordering* qui prenaient 25% du temps et qui naïvement pouvaient obtenir environ un facteur 10.

Nous avons constaté que plus les images étaient grandes, plus les gains étaient importants. On a donc simplement effectué les calculs suivants, avec T temps total de calcul :

– Image de 128*128 :

$$T_{gpu} = \frac{1}{2} * (\frac{T_{cpu}}{2}) + \frac{1}{5} * (\frac{T_{cpu}}{4}) + \frac{1}{20} * (\frac{T_{cpu}}{4}) \tag{3.1}$$

$$T_{gpu} = \frac{25}{80} T_{cpu} \approx \frac{1}{3} T_{cpu} \tag{3.2}$$

– Image de 2048*2048 :

$$T_{gpu} = \frac{1}{5} * (\frac{T_{cpu}}{2}) + \frac{1}{10} * (\frac{T_{cpu}}{4}) + \frac{1}{30} * (\frac{T_{cpu}}{4}) \tag{3.3}$$

$$T_{gpu} = \frac{49}{360} T_{cpu} \approx \frac{1}{7} T_{cpu} \tag{3.4}$$

Dans le cas d'une image 128x128 on peut donc espérer un gain de facteur 3 alors que pour une image 2048x2048 un gain de facteur 7. Nous verrons que nos estimations étaient très proches de la réalité à la suite.

3.7.2 Gains finaux sur l'algorithme d'*inpainting*

Depuis le début nous savions que nous n'obtiendrions pas un gain de facteur à 3 chiffres. Le fait que la *FFT* prenne en moyenne 50% du temps total et ne soit pas simplement optimisable n'est pas très efficace car les optimisations des nombreuses fonctions annexes ne permettent pas des gains exceptionnels. La figure 3.10 donne les résultats finaux et compare les facteurs d'accélération entre le GPU et les implémentations de *DCT* dont nous avons parlé, celle de FFTW3 et la notre.

On peut faire deux principales observations sur la figure 3.10. La *FFT* n'est pas très bien adaptée au GPU. Les calculs de *FFT* prennent environ 50% du temps total et cela se ressent. En revanche, il est trés intéressant de pouvoir comparer notre implémentation de *DCT*. Les gains sont compris entre un facteur 10 et 30. L'ensemble de notre algorithme n'exécute que peu de calcul. On peut donc considérer que notre algorithme est dépendant directement de la bande passante mémoire. Il reste encore plusieurs optimisations possibles sur cet algorithme. Néanmoins, nous avons fait la majeure partie du portage en incluant les optimisations. Cela a prit du temps, de longues heures de réflexions et beaucoup de café. CUDA n'est pas un outil simple à manipuler.

En pratique, nous avions espéré que si les gains étaient suffisant, nous pourrions créer une libraire qui serait mise à disposition des astrophyisiciens. Les gains sont un peu juste pour cet algorithme, mais il sont suffisamment encourageants pour continuer la recherche sur d'autres algorithmes de traitement d'image.

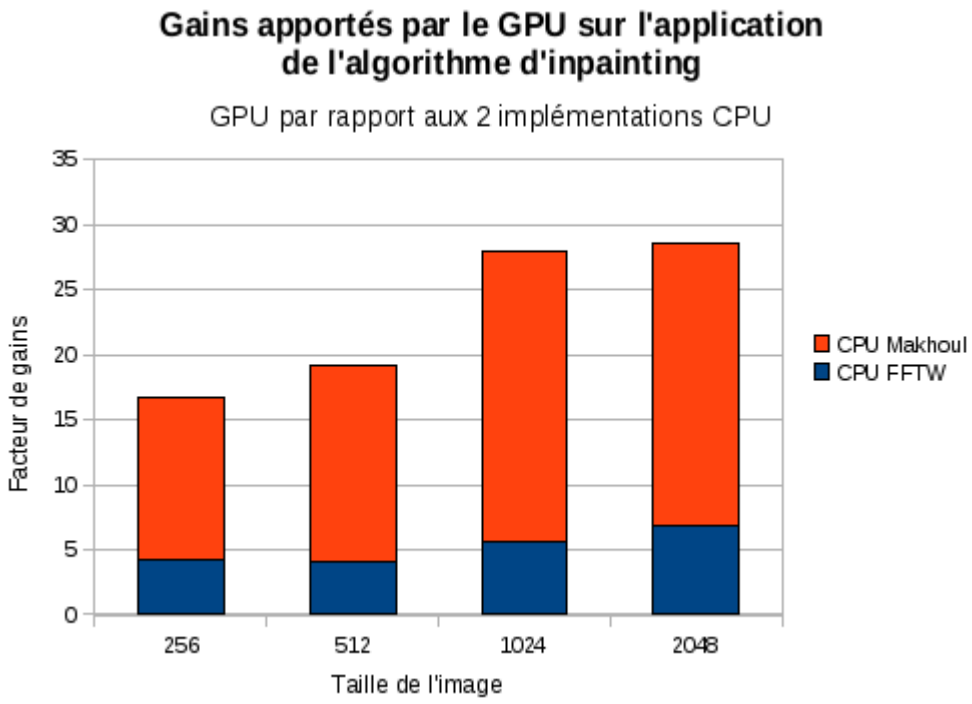


FIG. 3.10: Ce graphique représente les facteurs d'accélération apportés par le GPU. Les codes CPU utilisent pour **CPU Makhoul** notre implémentation de l'algorithme de Makhoul et pour **CPU FFTW**, la *DCT* de FFTW3. Le code CUDA utilise les permutations de Makhoul [11] ainsi que la *FFT* de CUDA (CUFFT). Pour les images en 512x512, nous utilisons notre FFT basée sur le code de V.Volkov. Les gains sont relativement faibles lorsque l'on compare á la *DCT* de FFTW3. En revanche les facteurs d'accélération par rapport à notre *DCT* sont compris entre 10 et 30.

Chapitre 4

Conclusion et Perspectives

Conclusion

Nous avons essayé au long de ce rapport de présenter au mieux l'API CUDA et plus particulièrement d'évaluer les possibilités de calcul des GPU Nvidia dans le cadre du traitement d'image. Tout d'abord nous avons décortiqué l'architecture des GPU et détaillé les particularités de CUDA. Sous forme d'une simple extension au langage C/C++, CUDA peut être vu comme une abstraction au hardware. Ce GPU est un ensemble de partitions composées de processeurs scalaires, allant jusqu'a 240 processeurs pour les dernières générations (10x24). La puce fonctionne de manière SIMT avec une synchronisation au niveau de 32 *threads* en parallèle. Il est possible de développer sans respecter cette méthode, mais ce sera au prix de pertes de performances importantes. Dans le but d'obtenir les meilleures performances possibles, il est réellement nécessaire d'avoir une bonne vision d'ensemble et simplement, de la pratique. Nvidia ne le cache pas, et présente en conférence une sommme de n termes, qui naïvement souffre de performances très moyennes, et qui après plusieurs étapes d'optimisations pas toujours simples, obtient un excellent rendement (cf. l'exemple *Reduction* [9]).

En seconde partie, nous avons présenté l'application de CUDA à un algorithme de traitement d'image : l'*inpainting*. Il est possible d'utiliser différentes transformées pour appliquer ce type d'algorithme. CUDA propose une librairie de Transformées de Fourier Rapide (CUFFT). Nous avons donc fait le choix d'implémenter une Transformée en Cosinus, car elle se calcule à partir de FFT. L'un des problèmes majeurs dans l'algorithme est que la FFT occupe entre 30 et 50% du temps de calcul ce qui rend les optimisations de chaques fonctions annexes très relatives, car chacunes n'occupent qu'au maximum 15%. Les sources de CUFFT ne sont pas disponibles, et reprendre une DCT dans son ensemble afin d'optimiser certaines parties de la FFT aurait été un travail intéressant mais que nous avons jugé trop long. Nous savions dès le départ que CUDA se prétait mieux à des algorithmes de types locaux, or ce n'est pas le cas pour la FFT. Mais il nous paraissait plus pertinent de tester un algorithme qui se prétait moins, autant pour constater les performances que pour voir s'il était possible d'optimiser dans un tel cas.

Nous avons comparé les résultats entre notre implémentation GPU et nos deux implémentation CPU (l'une utilisant la *FFT* de la librairie FFTW3 pour implémenter l'algorithme de Makhoul, et l'autre directement les *DCT* de la librairie FFTW3. Les résultats obtenus montrent que l'on obtient un gain sur GPU de facteur 2 à 8 selon les tailles d'images pour notre machine de test (équipée d'une Quadro FX4600 correspondant au haut de gamme de la génération de GPU sortie en 2007) et ce par rapport à l'implémentation de *DCT* de FFTW3. Pour compenser ces résultats en demie teinte, le facteur d'accélération par rapport à notre code de *DCT* se trouve entre 10 et 30 pour des images de 2048x2048. Ces résultats sont donc très encourageants et montrent que pour certains codes très spécifiques, on peut obtenir de très bonnes performances, même lorsque la quantité de calcul faible. Plus généralement, on peut dire que CUDA est très utile lorsque le calcul est intensif sur des données localisées, et lorsque l'on peut être SIMT.

Pour un code de production, un gain de facteur 8 commence à être exploitable. C'est là que les avantages de CUDA se révelent. Une machine standard pourvue d'un port PCI-Express x16 suffit. De plus, 1 seul coeur CPU est nécessaire pour lancer une application CUDA (1 processus CUDA par GPU est nécessaire), ce qui permet de ne pas totalement saturer la machine comme un code multithreadé pourrait le faire. CUDA s'intgre lentement dans divers domaines de l'informatique. De nombreux plugins accélérés avec CUDA sont développés pour Matlab. Plusieurs logiciels de traitement de vidéo proposent depuis peu l'utilisation de CUDA pour accélérer certains encodages. Si l'on ajoute à cela le rapport performances/puissance/prix, CUDA se révèle être un outil prometteur.

Perspectives

La suite de notre travail : transformation en ondelettes, ...

Les résultats encourageants que nous avons obtenus motivent le portage d'autres algorithmes de traitement d'image/signal avec le projet de constituer une librairie de fonctions. En particulier les ondelettes sont un outil incontournable dans ce domaine. Nous avons cherché si des travaux avaient été effectués sur différentes ondelettes. A part l'exemple d'ondelette de Haar disponibles dans le SDK de CUDA, nous n'avons rien trouvé. Il s'avère que le code est optimisé spécifiquement pour ces ondelettes qui ont des propriétés bien propres, ce qui empèche une adaptation généralisée à d'autres types d'ondelettes. C'est pourquoi nous avons choisi de nous orienter vers les ondelettes de Daubechies à 4 coefficients (DAUB4) qui sont elles, plus génériques.

L'intérêt que présentent ces ondelettes par rapport à une *FFT* est que le filtre est local à une petite zone de l'image. Avec l'expérience acquise durant ces 5 mois, nous espérons obtenir un résultat assez rapidement.

L'avenir pour CUDA

Nvidia travaille d'arrache pied sur CUDA. Durant mes 5 premiers mois de stage, j'ai pu travailler sur 3 versions différentes de CUDA, de la version 1.1 à la version 2.0. Chacune a apporté son lot de corrections, et les outils tels que le *profiler* ou l'*Occupancy Calculator*) ont été perfectionnés. Toutes ces évolutions permettent de rendre plus aisée les optimisations. qui sont indispensables et coûteuses en temps et ce, afin de tirer les meilleurs performances possibles du GPU.

L'arrivé des nouveaux GPU Nvidia GT-200 apporte maintenant le support du 64 bits en flottant pour CUDA. Ce point est critique dans le domaine de la recherche où la plupart les algorithmes déjà présents sont implémentés en double précision. De plus, ces GPU atteignent le téraflop avec une augmentation du nombre de processeurs scalaires et des fréquences.

L'évolution de CUDA peut lui permettre de voir un avenir proche assez favorable. Néanmoins, le monde du GPU Computing est bouillonant. Même si Nvidia semble actuellement avoir une petite avance sur ses concurrents, l'arrivée de technologies telle que AMD Fusion ou Intel Larrabee pourrait bien changer la donne. De plus, pour la sortie prochaine de DirectX 11, ATI devrait mettre à disposition une API standard très semblable à CUDA afin de rendre le GPGPU plus facile à prendre en main avec leur cartes, ce qui manquait jusque là.

Glossaire

DCT Transformée en Cosinus Discrète. 13–15, 17, 23, 24, 26, 28

FFT Transformée de Fourier Rapide. 2, 14–17, 19, 20, 22, 23, 26, 28

SDK Software Development Kit. 10

burst Accès mémoire en rafale qui permet d'accéder à des données rangées consécutivement en mémoire. 8

kernel Programme exécuté sur le GPU. 2, 5–7, 15, 16, 19–21, 23

thread Au sens de CUDA, un *thread* est representé par un bloc de code qui est executé sur 1 processeur "logique". Pour rapprel, le G80 en possède 128. 2, 5, 6, 8–12, 15–18, 20, 21, 26

warp Groupe de 32 threads d'un même bloc. Ils sont lancés en parallèle. 2, 5, 6, 9–11, 16, 18

ALU *Algorithmic Logic Unit*. 5

CUFFT Libraire de *FFT* de CUDA développée par Nvidia. 15, 20, 26

FFTW3 Libraire de *FFT* et *DCT* pour C standard. 14, 15, 22, 23, 26

nvcc Compilateur de CUDA. 16, 20

SIMT *Single Instruction Multiple Thread*. 26

Bibliographie

[1] Cfitsio library. http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html.

[2] CUFFT versus FFTW3 benchmark. http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/.

[3] Discrete Fourier Transform C library. http://www.fftw.org/.

[4] Healpix. http://healpix.jpl.nasa.gov/.

[5] *cudaMemset()* slowness. http://forums.nvidia.com/index.php?showtopic=72209&hl=cudamemset.

[6] P. Abrial, Y. Moudden, J.L. Starck, J. Bobin, M.J. Fadili, B. Afeyan, and M. Nguyen. Morphological component analysis and inpainting on the sphere : Application in physics and astrophysics. *Journal of Fourier Analysis and Applications*, 13(6) :729–748, 2007. special issue on "Analysis on the Sphere".

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, and Kayvon Fatahalian. Stanford University Graphics Lab, http://graphics.stanford.edu/projects/brookgpu/.

[8] Jean-Luc Danger. Fast fourier transform, Octobre 2001. http://comelec.enst.fr/~danger/amen_old/fft/.

[9] Mark Harris. AstroGPU 4 - Optimizing CUDA, 2007.

[10] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia Tesla : a Unified Graphics and Computing Architecture. IEEE Computer Society, Mars-Avril 2008.

[11] John Makhoul. A Fast Cosine Transform in One and Two Dimensions. In *IEEE Transactions Acoustits, Speech, and Signal Processing, VOL. ASSP-28, NO. I,*. IEEE, Février 1980.

[12] Damien Triolet. Nvidia cuda, aperï¿½u. Rapport technique, http://www.hardware.fr/articles/659-1/nvidia-cuda-apercu.html, Mars 2007.

[13] Vladimir Volkov. my speedy fft, Juin 2008. http://forums.nvidia.com/index.php?showtopic=69801&hl=speedy.

Annexe A

Codes

A.1 Transposées

A.1.1 Transposée du SDK de CUDA

```
#define BLOCK_DIM 16

__global__ void transpose(float *odata, float *idata, int width, int height)
{
        __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

        // read the matrix tile into shared memory
        unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
        unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
        if((xIndex < width) && (yIndex < height))
        {
                unsigned int index_in = yIndex * width + xIndex;
                block[threadIdx.y][threadIdx.x] = idata[index_in];
        }

        __syncthreads();

        // write the transposed matrix tile to global memory
        xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
        yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
        if((xIndex < height) && (yIndex < width))
        {
                unsigned int index_out = yIndex * height + xIndex;
                odata[index_out] = block[threadIdx.x][threadIdx.y];
        }
}
```

A.1.2 Adaptation à des donnï¿½es de type Complexes

J'ai implémenté cette transposée pour implémenter la version de FFT2D à l'aide du code V.Volkov.

```
#define BLOCK_DIM 16

__global__ void transpose(float2 *idata, float2 *odata, int width, int height)
{
        __shared__ float blockx[BLOCK_DIM][BLOCK_DIM+1];
        __shared__ float blocky[BLOCK_DIM][BLOCK_DIM+1];

        // read the matrix tile into shared memory
        unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
        unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
        if((xIndex < width) && (yIndex < height))
        {
                unsigned int index_in = yIndex * width + xIndex;
                blockx[threadIdx.y][threadIdx.x] = idata[index_in].x;
                blocky[threadIdx.y][threadIdx.x] = idata[index_in].y;
        }

        __syncthreads();

        // write the transposed matrix tile to global memory
        xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
        yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
        if((xIndex < height) && (yIndex < width))
        {
                unsigned int index_out = yIndex * height + xIndex;
                odata[index_out].x = blockx[threadIdx.x][threadIdx.y];
                odata[index_out].y = blocky[threadIdx.x][threadIdx.y];
        }
}
```

A.2 Ensemble des kernels utiliés par l'algorithme d'*inpainting* avec CUDA

A.2.1 En-tête du fichier de *kernels*

```
#ifndef _INPAINTING_CUDA_KERNEL_H_
#define _INPAINTING_CUDA_KERNEL_H_

#include <math.h>

#include "f2complex.cu"

#define BLOCK_DIM 16

#define BLOCK_DIM_X 128
#define BLOCK_DIM_Y 2

#define BLOCK_DIM_2   2
#define BLOCK_DIM_4   4
```

```
#define BLOCK_DIM_8   8
#define BLOCK_DIM_32  32
#define BLOCK_DIM_64  64
#define BLOCK_DIM_128 128
#define BLOCK_DIM_256 256
```

A.2.2 Mise à 0 d'un tableau

```
// Fill a matrix with 0s
__global__
void zeros_kernel(Real * data, int heigth)
{
  unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

  unsigned int index = xIndex + __umul24(blockIdx.y, heigth);

  data[index] = 0;
}
```

A.2.3 Soustraction de 2 tableaux

```
// Perform "-" operation on 2 matrix
__global__
void minus_kernel(Real * res, Real * data1,
  Real * data2, int heigth)
{
  unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

  unsigned int index = xIndex + __umul24(blockIdx.y, heigth);

  res[index] = data1[index] - data2[index];
}
```

A.2.4 Seuillage des coefficients d'un tableau

```
// Perform a*(abs(a) >= seuil)
__global__
void abs_kernel(Real * data, float seuil,
int heigth)
{
  unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

  unsigned int index = xIndex + __umul24(blockIdx.y, heigth);

  data[index] = data[index] * (fabs(data[index]) >= seuil);
}
```

A.2.5 Opération comprenant une addition et une multiplication de tableaux

```
// Perform a computation to init forward DCT
__global__
void init_fDCT2d_kernel(Real* alpha, Real* out,
Real* res, Real* mask,
int heigth)
{
  unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

  unsigned int index = xIndex + __umul24(blockIdx.y, heigth);

  alpha[index] = out[index] + mask[index] * res[index];
}
```

A.2.6 Preswap pré Transformée de Fourier Rapide

```
// preswap the 4 parts in one kernel
__device__
void preswap_all(Real* block, Real* out,
 int width, int heigth)
{
  int globalIn;
  int indexIn;
  int indexOut;

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int bx = blockIdx.x;
  int by = blockIdx.y;

  int bdx2 = blockDim.x>>1;



  if(ty==(2*(ty>>1)))
    {
      if(tx < bdx2){
globalIn = by*blockDim.y*width + ty*width + bx*blockDim.x + 2*tx;

indexIn = 2*tx + blockDim.x*ty;
indexOut = globalIn/2;

out[indexOut] = block[indexIn];
      }
      else{
globalIn = by*blockDim.y*width + ty*width + bx*blockDim.x + 2*(blockDim.x - tx) - 1;
```

```
indexIn = 2*(blockDim.x - tx) + ty*blockDim.x - 1;
indexOut = (globalIn - 1)/2 + (width-1) - 2*(blockDim.x - tx -1) - bx*blockDim.x;

out[indexOut] = block[indexIn];
//      printf("tx: %d, ty: %d, bx: %d, by: %d, in: %d, gin: %d, gout: %d\n",tx,ty,bx,by,i
      }
    }
  else
    {
      if(tx<bdx2){
globalIn = by*blockDim.y*width + ty*width + bx*blockDim.x + 2*tx;

indexIn = 2*tx + blockDim.x*ty;
  indexOut = (heigth*width - (ty+1)*width - by*blockDim.y*width) + globalIn/2 + width/2;

  out[indexOut] = block[indexIn];
      }
      else{
globalIn = by*blockDim.y*width + ty*width + bx*blockDim.x + 2*(blockDim.x - tx) - 1;

indexIn = 2*(blockDim.x - tx) + ty*blockDim.x - 1;
indexOut = (globalIn - 1)/2 + (heigth*width -(ty+1)*width - by*blockDim.y*width)
  + (width-1) +width/2 - 2*(blockDim.x - tx -1) - bx*blockDim.x;

out[indexOut] = block[indexIn];
      }
    }
}


/*
 * Swapping pre FFT.
 * Coalesced acces but bank conflicts ().
 */
__global__
void preswap_fDCT_kernel(Real* in, Real* out,
     int width, int heigth)
{
  // 1 array for each square
  __shared__ Real block[BLOCK_DIM_X*BLOCK_DIM_Y];

  int xIndex = blockIdx.x*blockDim.x + threadIdx.x;
  int yIndex = blockIdx.y*blockDim.y + threadIdx.y;

  int index = yIndex*width + xIndex;

  // copy to shared mem
  block[threadIdx.x + threadIdx.y*blockDim.x] = in[index];
```

```
  __syncthreads();


  preswap_all(block,out,width,heigth);
}
```

A.2.7 Recombinaison complexe post *FFT*

```
__global__
void complexSwap_v1_fDCT_kernel(Complex * in, Real * out,
int width, int heigth)
{
  unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;

  // Inverse indexes to have coalesced read/writes
  int j = xIndex;
  int i = blockIdx.y;


  Complex a,b,ab,adivb;
  float real, real2, imag, imag2;

  a.x = 0;
  a.y = -M_PI*i/2/width;

  a = c_exp(a);

  b.x = 0;
  b.y = -M_PI*j/2/heigth;

  b = c_exp(b);

  ab = c_mul(a,b);
  adivb = c_div(a,b);

  real = in[j + width*i].x;
  imag = in[j + width*i].y;


  if (j==0) {
    out[width*i] = 0.5*(real *a.x - imag*a.y
+ real*adivb.x - imag*adivb.y);
  }else{
    real2 = in[width-j+width*i].x;
    imag2 = in[width-j+width*i].y;

    out[j+width*i] = 0.5*(real*ab.x - imag*ab.y
  + real2*adivb.x - imag2*adivb.y);
  }
```

```
  /*
    // "optimized version"
    // Less uncoalesced access, less divergence, less good perf.. Why?
    // No idea atm...
  if(i==0){
    real2 =  in[j].x;
    imag2 = -in[j].y;
  }else {
    real2 =  in[j + (width*heigth-width*i)].x;
    imag2 = -in[j + (width*heigth-width*i)].y;
  }

  out[j+width*i] = 0.5*(real*ab.x - imag*ab.y
+ real2*adivb.x - imag2*adivb.y);
  */
}
```

A.2.8 Recombinaison complexe pre *FFT* inverse

```
__global__
void complexSwap_v1_iDCT_kernel(Real *in, Complex * out,
    int width, int heigth)
{
  unsigned int xIndex = blockIdx.x * BLOCK_DIM_64 + threadIdx.x;

  // TEMPORARY SOLUTION ...
  int j = xIndex;
  int i = blockIdx.y;

  Complex a, b;

  Complex tmp;

  unsigned int index = j + i*width;

  // special index
  if(index == 0){
    out[0].x = in[0];
    out[0].y = 0; // ?????????????????????????? A VERIFIER.
  }

  // 1st line
  if(i==0 && j>0){
    a.x = 0;
    a.y = M_PI_2/heigth*j;

    a = c_exp(a);

    b.x =  in[index];
```

```
    b.y = -in[width-index];

    b = c_mul(a,b);

    out[index].x = b.x;
    out[index].y = b.y;
  }

  // 1st column
  if(i>0 && i<width && j==0){
    b.x = 0;
    b.y = M_PI_2/width*i;

    b = c_exp(b);

    a.x =  in[heigth*i];
    a.y = -in[heigth*(width-i)];

    a = c_mul(a,b);

    out[heigth*i].x = a.x;
    out[heigth*i].y = a.y;
  }


  // the remaining matrix
  if(i>0 && j>0){
    a.x = 0;
    a.y = M_PI_2*i/width;

    b.x = 0;
    b.y = M_PI_2*j/heigth;

    a = c_exp(a);
    b = c_exp(b);

    tmp.x =
      in[j+heigth*i] - in[heigth-j+heigth*(width-i)];
    tmp.y =
      -(in[j+heigth*(width-i)] + in[heigth-j +heigth*i]);

    a = c_mul(tmp, c_mul(a,b));
    //    if((i==1 || i==heigth-1) && (j==1 || j== heigth-1))
      //printf("%f -- %f --- %f -- %f -- %f -- %f\n",a.x, a.y, in[j+heigth*i], in[heigth-j
    out[j + heigth*i].x = a.x;
    out[j + heigth*i].y = a.y;
  }
}
```

A.2.9 Permutations post *FFT*

```
__global__
void postswap_v2_kernel(Real * in, Real * out, int w, int h)
{

  __shared__ Real block[BLOCK_DIM_X];

  int tx;
  tx = threadIdx.x;

  int bdx2 = BLOCK_DIM_X/2;

  int normalize = w*h;

  // LIGNE SORTIE PAIRE
  if(blockIdx.y<(h/2)){
    if(tx < bdx2){
      int indexIn    = tx + blockIdx.x*bdx2 + blockIdx.y*w;
      int indexBlock = 2*tx;

      block[indexBlock] = in[indexIn];
    }else{
      int indexIn    = w - (bdx2) + tx-bdx2 - blockIdx.x*(bdx2) + blockIdx.y*w;
      int indexBlock = 2*bdx2-1 - 2*(tx-bdx2);

      block[indexBlock] = in[indexIn];
    }
  }
  // LIGNE SORTIE IMPAIRE
  else{
    if(tx < bdx2){
      int indexIn    = tx + blockIdx.x*bdx2 + blockIdx.y*w;
      int indexBlock = 2*tx;

      block[indexBlock] = in[indexIn];
    }else{
      int indexIn    = w - (bdx2) + tx-bdx2 - blockIdx.x*(bdx2) + blockIdx.y*w;
      int indexBlock = 2*bdx2-1 - 2*(tx-bdx2);

      block[indexBlock] = in[indexIn];
    }
  }
  // FIN COPIE EN SMEM

  __syncthreads(); // OR NOT ???

  int indexOut;
  int indexBlock;
```

```
  if(blockIdx.y<(h/2)){     // LIGNES PAIRES
    indexBlock = tx;
    indexOut   = (2*blockIdx.y)*w + blockIdx.x*2*bdx2 + tx;

    out[indexOut] = block[indexBlock]/normalize;
  }else{                    // LIGNES IMPAIRES
    indexBlock = tx;
    indexOut   = (2*h-1 -2*blockIdx.y)*w + blockIdx.x*2*bdx2 + tx;

    out[indexOut] = block[indexBlock]/normalize;
  }
}
```

A.2.10 *Kernels* utilitaires

```
__global__
void real2complex_kernel(Real* in, Complex* out,
     int width, int heigth)
{
  unsigned int xIndex = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;

  unsigned int index = __umul24(blockIdx.y, width) + xIndex;

  float2 a;

  a.x = in[index];
  a.y = 0;

  out[index] = a;
}


// Kernel used to get the real part of Complex array
// to make Real array.
// Coalesced but bank conflicts
__global__
void complex2real_kernel(Complex* in, Real* out,
     int width, int heigth)
{
  int xIndex = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;

  int index = __umul24(blockIdx.y, width) + xIndex;

  // Volatile to force register usage
  volatile float2 a;

  a.x = in[index].x;
  a.y = in[index].y;
```

```
  out[index] = a.x;
}
#endif // _INPAINTING_CUDA_KERNEL_H_
```

Annexe B

Illustrations

B.1 Application de l'algorithme d'*inpainting* sur une image de la Sphère

La figure B.1 représente l'image d'entrée, le masque et l'image traitée par l'algorithme d'*inpainting*. L'image à été décomposée en 12 sous images pour qu'elle puisse être traitée. La décomposition se fait à l'aide de la librairie Healpix.

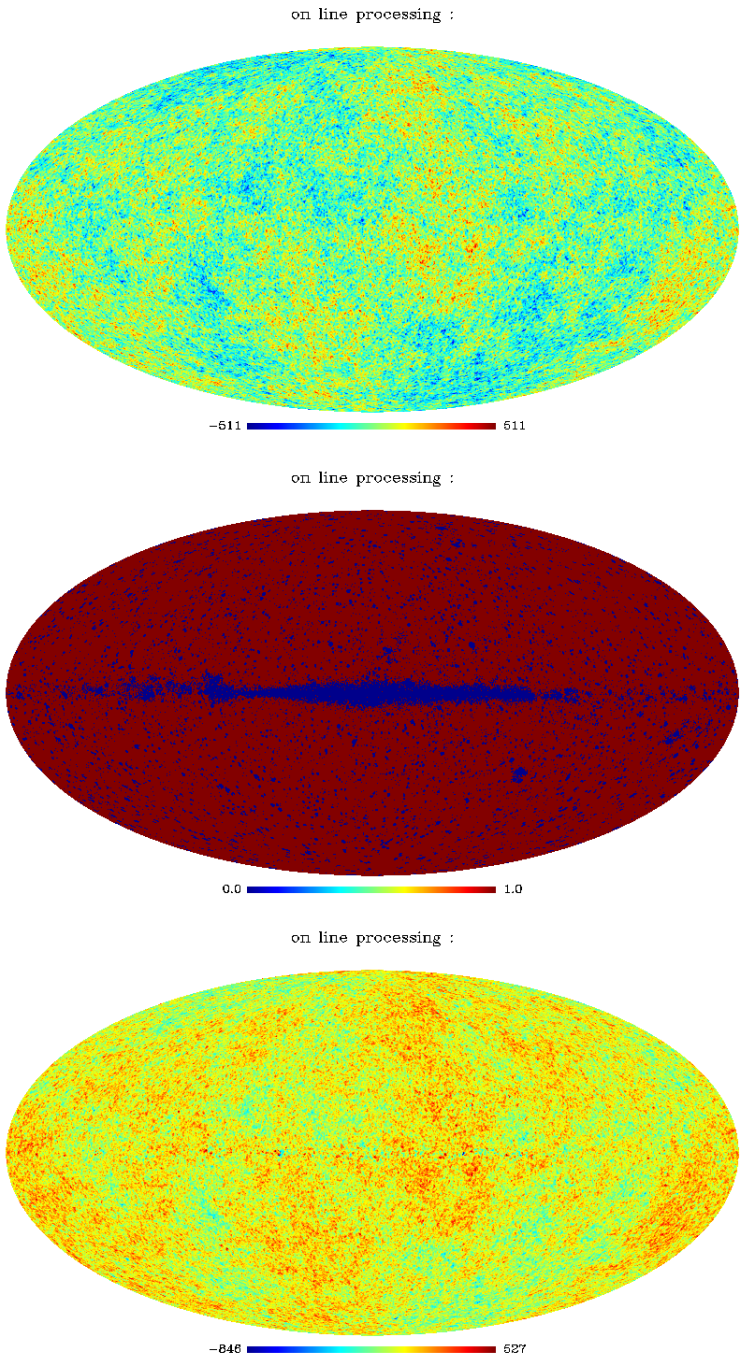


FIG. B.1: *Inpainting* avec CUDA sur le Fond Diffus Cosmologique. On note que l'algorithme a quelques problèmes en certains points. Ces points sont sûrement des jonctions entre les images découpées