Institut National des Sciences Appliquées de Toulouse

Département de Génie Mathématiques et Modélisation

Projet 5A
THALES Alenia Space

Tuteurs de projet : RONDEPIERRE Aude, WEISS Pierre

Algorithmes pour le débruitage et la déconvolution d'images en résolution parallèle avec CUDA

DUMAS Loïc, TAPIA-PIQUEREY Benoît
<ldumas@etud.insa-toulouse.fr, btapia_p@etud.insa-toulouse.fr>

Toulouse, Janvier 2010

Table des matières

1 Résumé

Le débruitage et la déconvolution d'image sont des opérations lourdes et coûteuses en temps de calculs. Le but de ce projet est d'étudier des méthodes de traitement d'image, et de développer les codes de calculs correspondants, d'abord en Matlab pour tester la théorie, puis en CUDA pour des calculs ultra rapides.

CUDA est une technologie de NVIDIA qui permet d'exploiter la puissance des GPU (Graphic Processor Unit) et leurs capacités de calculs parallèles pour diminuer les temps de calculs. CUDA est une API pour le langage C, pour plus de renseignements : `http://www.nvidia.fr/object/cuda_what_is_fr.html`.

2 CUDA : Une nouvelle architecture pour la programmation sur carte graphique

Dans ce chapitre nous allons présenter de façon succincte les notions de bases de l'architecture et de la programmation CUDA. La connaissance de toutes ces informations n'est pas nécessaire dans son intégralité pour le lecteur souhaitant réaliser un premier programme via CUDA. De la même façon, toutes les notions présentées dans la suite ne sont pas suffisantes pour faire du lecteur, s'il débute et ne possède aucune connaissance préalable de CUDA, un programmeur de haut niveau. Ce chapitre peut en revanche tout à fait suffire à modifier ou créer un programme incluant du langage CUDA pour obtenir une accélération importante en termes de temps de calcul. Les notions n'ayant pas été utiles lors de la programmation de nos algorithmes ne seront donc que brièvement décrites ; juste assez pour permettre au lecteur qui découvre CUDA de prendre connaissances de ces fonctionnalités, pour éventuellement les intégrer à ses futurs codes après des recherches plus approfondies.

2.1 Premières notions

2.1.1 Introduction

CUDA (Compute Unified Device Architecture) est un modèle de programmation parallèle qui s'apparenterait presque à un logiciel permettant de manipuler de façon transparente la puissance d'une carte graphique composée d'une vingtaine de multiprocesseurs, tout en étant facile de prise en main pour des utilisateurs familiers du langage C.

Trois notions essentielles sont à la base de CUDA : une hiérarchie de groupes de *threads*, des mémoires partagées et une barrière de synchronisation. Celles-ci sont mises à la disposition du programmeur à travers seulement quelques modifications, ou plutôt extensions, du langage C.

Ces notions donnent la possibilité au programmeur de gérer avec précision le parallélisme offert par la carte graphique. Elles permettent à celui-ci de partitionner le problème en sous problèmes pouvant être résolus de manière indépendante et parallèle, puis en parties encore plus fines pouvant, elles, être résolues de manière coopérative et parallèle. Un programme CUDA peut alors s'exécuter sur un certain nombre de processeurs, le tout sans que le programmeur n'ait conscience de l'architecture de sa carte graphique.

Les extensions du langage C apportées par CUDA permettent au programmeur de définir des fonctions appelées *kernels*, qui se dérouleront sur carte graphique, en

parallèle. Ces fonctions seront donc exécutées N fois par N CUDA *threads* différents.

Un *kernel* est déclaré comme une simple fonction C, à l'exception faite du préfixe : **__global__**. D'autre part, pour chaque appel au *kernel*, le nombre de *threads* sera spécifié en utilisant la syntaxe <<< ... >>> :

```
// definition du Kernel
__global__ void vecAdd(float* A, float* B, float* C)
{
}
int main()
{...
// appel au kernel
vecAdd<<<1, N>>>(A, B, C);
}
```

Chaque *thread*, exécutant le code d'un *kernel*, se voit attribuer un identifiant à travers la variable standard **threadIdx**. Cet identifiant est accessible à l'intérieur du *kernel*. A titre d'exemple, le code suivant réalise l'addition de deux vecteurs A et B, de dimension N, et place le résultat dans le vecteur C :

```
__global__ void vecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
// appel au kernel
vecAdd<<<1, N>>>(A, B, C);
}
```

2.1.2 Les *threads*

La variable standard **threadIdx**, invoquée dans le paragraphe précédent, est un vecteur à trois composantes. Par conséquent, les *threads* peuvent être identifiés à travers une dimension, deux dimensions, ou trois dimensions formant ainsi des *blocks* d'une dimension, de deux dimensions ou de trois dimensions. Cette caractéristique prend tout son sens dès lors que l'on manipule des tenseurs d'ordre 2 ou 3. Par exemple, le *kernel* suivant réalise l'addition de la matrice A par B, deux matrices de même dimension NxN, et place le résultat dans la matrice C :

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N]){
int i = threadIdx.x;
int j = threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
}
int main(){
// appel au kernel
dim3 dimBlock(N, N);
matAdd<<<1, dimBlock>>>(A, B, C);
```

```
10 }
```

L'indice d'un *thread* et son identifiant sont liés de la façon suivante :
- Pour un *block* d'une dimension l'indice et l'identifiant sont les mêmes
- Pour un *block* de deux dimensions de taille (Dx, Dy), l'identifiant du *thread* d'indice (x, y) est (x+y*Dx)
- Pour un *block* de trois dimensions de taille (Dx,Dy,Dz), l'identifiant du *thread* d'indice (x,y,z) est (x+y*Dx+z*Dy)

Les *threads* d'un même *block* peuvent communiquer ensembles en partageant leurs données via la mémoire partagée et se synchroniser pour coordonner leurs accès mémoire. Concrètement, le programmeur peut ordonner une synchronisation des *threads* en employant l'instruction **___syncthreads()**. Cette fonction intrinsèque agit comme une barrière pour les *threads* d'un même *block*, barrière qu'aucun d'eux ne pourra franchir seul. En d'autres termes un *thread* rencontrant cette instruction devra attendre que tous les autres *threads* l'aient atteinte avant de poursuivre.

Pour fonctionner de manière efficace, la mémoire partagée est supposée rapide d'accès, **___syncthreads()** est supposée ne pas être coûteuse et tous les *threads* d'un *block* doivent résider sur le même processeur. Par conséquent, le nombre de *threads* par *block* est restreint par les ressources mémoire limitées d'un processeur. Sur une architecture NVIDIA Tesla, un *block* peut contenir jusqu'à 512 *threads*.

Cependant, un *kernel* peut être exécuté par plusieurs *blocks* de même dimension. Dans ce cas le nombre de *threads* est bien sûr le résultat du produit du nombre de *threads* par *block*, par le nombre de *blocks*. Ces *blocks* sont disposés à l'intérieur d'une *grid* (grille) de dimension strictement inférieure à trois (voir Figure suivante). La dimension de cette grille peut être spécifiée via le premier argument entre les caractères $<<< \cdots >>>$. Là encore les *blocks* peuvent être identifiés à l'intérieur d'un *kernel* grâce à la variable standard **blockIdx**. La dimension d'un *block* peut, quand à elle, être accessible à travers la variable standard **blockDim** (elle correspond au nombre de *threads* présent dans un *block*).

Le code précédent devient alors :

```
1  __global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
2  {
3  int i = blockIdx.x * blockDim.x + threadIdx.x;
4  int j = blockIdx.y * blockDim.y + threadIdx.y;
5  if (i < N && j < N)
6  C[i][j] = A[i][j] + B[i][j];
7  }
8  int main()
9  {
10 // appel au kernel
```

```
11 dim3 dimBlock(16, 16);
12 dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,(N + dimBlock.y - 1) / dimBlock.
       y);
13 matAdd<<<dimGrid, dimBlock>>>(A, B, C);
14 }
```

La dimension du *block* est ici choisie arbitrairement. La dimension de la grille est telle qu'il y ait suffisamment de *blocks* pour avoir un *thread* par composante comme précédemment.

Les *blocks* s'exécutent de façon indépendante sans ordre particulier. Le nombre de *blocks* par grille ne dépend pas du nombre de processeurs disponibles mais des données à manipuler. En effet sur une GeForce GTX 260, 192 processeurs sont disponibles alors que le nombre maximal de *blocks* par grille est de 65535.

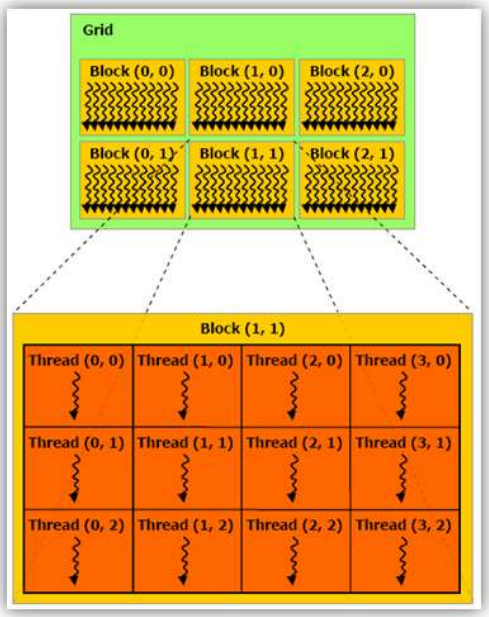

FIG. 1 – Illustration d'une grille

2.1.3 Les mémoires

Les *threads* ont accès à des données se situant dans des mémoires distinctes. Chaque *thread* possède sa propre mémoire appelée *localmemory* (mémoire locale). Les *blocks* eux, possèdent une *sharedmemory* (mémoire partagée) accessible uniquement par les *threads* du *block* en question. Enfin, tous les *threads* ont accès aux données se trouvant dans la *globalmemory* (mémoire globale).

Il existe également deux autres mémoires, à savoir la *constantmemory* (mémoire constante) et la *texturememory* (mémoire de texture).

2.1.4 *Host* et *Device*

Comme illustré sur la figure suivante, les *threads* s'exécutent sur une machine physique (le *device*) séparée, en concurrence avec le reste du code C se déroulant sur ce que l'on appelle le *host*. Ces deux entités possèdent leur propre DRAM (Dynamic Random Access Memory) appelées respectivement *devicememory* et *hostmemory*. Un programme CUDA doit donc manipuler les données de sorte qu'elles soient copiées sur le *device* pour y être traitées, avant d'être recopiées sur le *host*.

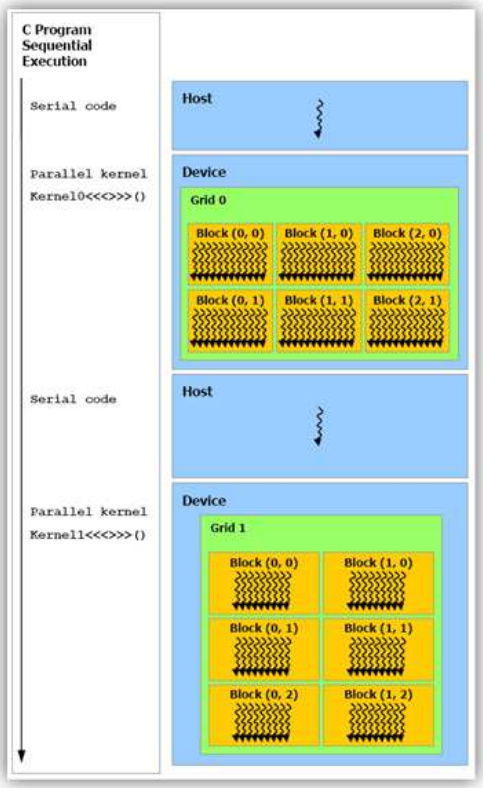

FIG. 2 – Echange *host*/*device*

2.2 Architecture du GPU

L'architecture Tesla datant de novembre 2006, s'appuie sur des multiprocesseurs SIMT avec mémoire partagée embarquée.

Lorsqu'un programme CUDA invoque un *kernel* depuis le *host*, les *blocks* de la grille sont énumérés et distribués aux multiprocesseurs ayant suffisamment de ressources pour les exécuter. Tous les *threads* d'un *block* sont exécutés simultanément sur le même multiprocesseur. Dès lors que tous les *threads* d'un *block* ont terminé leurs instructions, un nouveau *block* est lancé sur le multiprocesseur.

Un multiprocesseur est composé de huit processeurs. La combinaison d'une barrière de synchronisation **___syncthread()** à moindre coût, d'un procédé de création de *threads* rapide, et l'absence de planification concernant l'ordre d'exécution des *threads* autorise au programmeur de coder avec une faible granularité en assignant par exemple une seule composante d'un vecteur à un *thread*, ou encore un pixel d'image...

Pour manipuler des centaines de *threads*, le multiprocesseur emploie une nouvelle architecture appelée SIMT pour Single-Instruction, Multiple Threads (une seule instruction, plusieurs *threads*). C'est donc l'unité SIMT d'un multiprocesseur qui crée, planifie, et exécute les *threads* par groupe de 32. Ces groupes de *threads* sont appelés *warps*, un $half - warp$ étant soit la première moitié d'un *warp*, soit la seconde. Les *threads* à l'intérieur d'un *warp* commencent leur exécution à la même instruction dans le programme mais sont libres de s'exécuter indépendamment.

Lorsqu'un multiprocesseur se voit offert un ou plusieurs *blocks* de *threads* à exécuter, il les divise en *warps* dont l'exécution sera planifiée par l'unité SIMT. La méthode employée pour diviser les *blocks* est toujours la même : le premier *warp* contient le *thread* d'identifiant 0, et les *threads* consécutifs jusqu'au *thread* d'identifiant 31, puis le second *warp* contient les 32 *threads* suivants et ainsi de suite... (L'indexage des *threads* est détaillé au chapitre 2.1.2).

Dès lors qu'un *warp* prêt à être exécuté est sélectionné par l'unité SIMT, l'instruction suivante est lancée pour tous les *threads* actifs du *warp* (certaines instructions de contrôle peuvent rendre inactifs des *threads* à l'intérieur d'un *warp*). Une seule instruction à la fois est réalisée par *warp*, autrement dit les performances seront maximales si les 32 *threads* composant le *warp* sont d'accord sur le chemin à prendre. C'est pourquoi il est préférable lors des structures de contrôle de ne pas faire diverger les *threads* d'un *warp*. Si toutefois cette situation apparaît, le *warp* exécutera alors tous les chemins différents pris par les *threads* le composant, et ce en série, chemin après chemin. Une fois que tous les chemins sont terminés, les *threads* convergent à nouveau vers le même chemin.

Comme le montre la figure ci-dessous, la mémoire personnelle d'un multiproces-

seur est organisée de la sorte :
- Un banc de 32 registres par processeur
- Un bloc mémoire commun à tous les processeurs, où se situe la mémoire partagée
- Une mémoire cache en lecture seule, conçue pour accélérer l'accès aux données présentes dans la mémoire constante (qui est également une mémoire en lecture seule), partagée entre les processeurs
- Une mémoire cache en lecture seule, conçue pour accélérer l'accès aux données présentes dans la mémoire de texture (qui est également une mémoire en lecture seule), partagée entre les processeurs

A noter qu'il est possible de lire et écrire dans les mémoires dites globale et locale de la mémoire du *device*, et qu'il n'existe pas de mémoire cache permettant d'accélérer l'accès aux données présentent dans les mémoires globale et locale.

Le nombre de *block* pouvant être manipulés en simultané par un multiprocesseur -encore appelé nombre de *blocks* actifs par multiprocesseur- dépend des ressources disponibles pour ce multiprocesseur. En d'autres termes, le nombre de registres nécessaires par *thread*, mais aussi la taille de la mémoire partagée utilisée par *block*, le tout pour un *kernel* donné, vont avoir un impact direct sur le nombre de *block* maximum pouvant être gérés par multiprocesseur. Si d'aventure il n'y avait pas suffisamment de ressources disponibles par multiprocesseur, l'exécution du *kernel* échouerait. Le nombre maximum de *blocks* actifs par multiprocesseur, ainsi que le nombre maximum de *warps* actifs et le nombre maximum de *threads* actifs sont donnés en Annexe (section B).

S'il arrive que dans un même *warp*, plusieurs *threads* souhaitent écrire à la même adresse en mémoire (que ce soit dans la mémoire globale ou la mémoire partagée), aucune garantie n'est apportée concernant ni le nombre effectif d'écritures (encore qu'un des *threads* au moins soit certain d'y avoir accès), ni même concernant l'ordre dans lequel ils y écriront. Pour faire face à ce problème majeur, il est possible d'utiliser des fonctions dites atomiques. Ces fonctions garantissent un accès exclusif à chaque *thread* sans toutefois qu'un ordre soit défini entre les *threads* pour l'écriture à l'adresse en question. Petit bémol tout de même, ces fonctions atomiques ne sont pas supportées par toutes les cartes graphiques, cela dépend de la *computecapability* de la carte en question (voir détails en Annexe, section B).

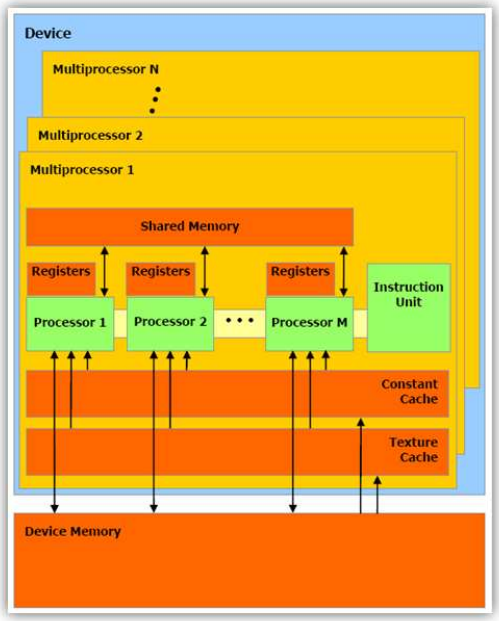

FIG. 3 – Multiprocesseur SIMT avec mémoire partagée embarquée

2.3 L'interface de programmation CUDA

2.3.1 Une extension de langage C

Le but annoncé de l'interface de programmation CUDA, est de permettre à un programmeur habitué à coder en C, de pouvoir se lancer dans la programmation sur carte graphique sans trop de difficulté.

On distinguera quatre éléments clefs comme étant à la base des extensions apportées au langage C :
- Les types permettant de définir l'entité par qui une fonction doit être exécutée
- Les types permettant de définir la mémoire où réside une variable donnée
- Des caractères particuliers pour définir la grille lors de l'exécution d'un *kernel* à partir du *host*
- Des variables standards faisant références aux indices et dimensions des paramètres de la grille

Chaque fichier source comportant l'une de ces extensions devra être compilé à l'aide du compilateur **nvcc**. Chacune de ses extensions est cependant accompagnée de restrictions qui seront parfois signalées lors de la compilation, par le compilateur nvcc.

2.3.1.a Les types qualifiant les fonctions

Le préfixe ___**device**___ permet de déclarer une fonction :
- Exécutée sur le *device*
- Appelable uniquement par le *device*

Le préfixe ___**global**___ permet de déclarer un *kernel*. Une telle fonction est :
- Exécutée sur le *device*
- Appelable uniquement par le *host*

Le préfixe ___**host**___ permet de déclarer une fonction :
- Exécutée sur le *host*
- Appelable uniquement par le *host*

A noter que le préfixe ___**host**___ est équivalent à l'absence de préfixe. D'autre part il est envisageable d'utiliser les préfixes ___**host**___ et ___**device**___ simultanément, auquel cas la fonction est compilée à la fois pour le *host* et le *device*. Enfin l'appel à une fonction ___**global**___ est asynchrone. L'instruction suivante peut être traitée par le *host* avant que le *device* n'ait terminé l'exécution du *kernel*.

Certaines restrictions sont toutefois à prendre en compte :
- Une fonction déclarée avec pour préfixe ___**device**___ ou ___**global**___ ne peut être récursive
- L'utilisation de variables statiques est interdite à l'intérieur d'une fonction ___device___ ou ___**global**___
- Les fonctions ___**device**___ ou ___**global**___ ne peuvent avoir un nombre variable d'arguments
- Il n'est pas possible d'utiliser des pointeurs sur des fonctions contenant l'adresse d'une fonction ___**device**___
- La combinaison des préfixes ___**host**___ et ___**global**___ n'est pas permise
- Les fonctions ___**global**___ doivent être de type void
- Tout appel à une fonction ___**global**___ doit être accompagné d'une configuration pour la grille (cf. section 2.3.1.c)
- Les arguments d'appel à une fonction ___**global**___ sont transmis à travers la mémoire partagée, et sont limités à 256 octets.

2.3.1.b Les types qualifiant les variables

Le préfixe ___**device**___, s'il est utilisé sans aucun autre préfixe, qualifie une variable qui : - Réside dans la mémoire globale - Jouit d'une durée de vie égale à celle de l'application - Est accessible par tous les *threads* d'une grille, et par le *host*

via la runtime library, que nous pouvons traduire par bibliothèque d'exécution et qui sera présentée plus loin (cf. section 2.3.2)

Le préfixe **___constant___** , pouvant éventuellement être combiné avec le préfixe **___device___**, déclare une variable qui : - Réside dans la mémoire constante - Jouit d'une durée de vie égale à celle de l'application - Est accessible par tous les *threads* d'une grille, et par le *host* via la bibliothèque d'exécution

Le préfixe **___shared___**, pouvant éventuellement être combiné avec le préfixe **___device___**, déclare une variable qui : - Réside dans la mémoire partagée d'un *block* de *threads* - Jouit d'une durée de vie égale à celle du *block* - Est uniquement accessible par les *threads* du *block*

A noter qu'il n'est garanti que tous les *threads* d'un *block* aient accès à une variable résidant dans la mémoire partagée, qu'après l'instruction **___syncthreads()**. Il n'est pas possible de déclarer un tableau de taille dynamique (connue uniquement lors de l'exécution) à l'intérieur d'un *kernel*. Pour ce faire il est nécessaire de déclarer une telle variable comme étant externe :

```
1 extern __shared__ float shared[];
```

Toutes les variables déclarées de la sorte commencent à la même adresse en mémoire, par conséquent si plusieurs tableaux veulent être utilisés il faut manipuler des offsets. Déclarer les variables suivantes...

```
1 short array0[128];
2 float array1[64];
3 int   array2[256];
```

...pour qu'elles soient en mémoire partagée et de taille dynamique, devra se faire de la façon suivante :

```
1 extern __shared__ char array[];
2 __device__ void fonc() // fonction __device__ ou __global__
3 {
4 short* array0 = (short*) array;
5 float* array1 = (float*)&array0[128];
6 int*   array2 = (int*)  &array1[64];
7 }
```

Là encore certaines restrictions sont en prendre en considération :
- Ces qualificatifs ne peuvent être utilisés sur des variables déclarées avec les mots clefs **struct** ou **union**, ni même pour des variables locales à l'intérieur d'une fonction compilée pour le *host*
- La valeur des variables résidant dans la mémoire constante ne peut être assignée qu'à partir du *host* via certaines fonctions que nous présenterons plus loin (cf. section 2.3.4.c)
- Les variables se trouvant dans la mémoire partagée ne peuvent se voir assigner une valeur lors de leur déclaration

Les variables dites automatiques, déclarées à l'intérieur d'une partie du code développé pour le *device* et sans aucun des préfixes cités, résident dans les registres. Il est cependant possible que ces variables soient redirigées vers la mémoire locale, ce qui entraînerait une baisse importante des performances (cf. section 2.4.3.b).

L'adresse d'une variable **___contant___**, **___shared___** ou **___device___**, ne peut être utilisée qu'à partir du *device*. En revanche l'adresse d'une variable **___device___** ou **___constant___** obtenue à l'aide de la fonction **cudaGetSymbolAddress()** (cf. section 2.3.4.c) ne peut être utilisée qu'à partir du *host*.

2.3.1.c Configuration de la grille

Comme précisé précédemment, tout appel à une fonction **___global___**, doit spécifier la configuration de la grille pour le *kernel* en question.

Cette configuration définit la dimension de la grille et des *blocks* utilisés lors de l'exécution. Elle est spécifiée en employant une expression de la forme <<<**Dg, Db, Ns, S** >>>, entre le nom du *kernel* et les arguments de celui-ci, où :
- **Dg** est de type **dim3** (cf. section 2.3.2.b) et spécifie la dimension de la grille de sorte que **Dg.x * Dg.y** représentent le nombre de *blocks* (**Dg.z** doit être égal à 1)
- **Db** est de type **dim3** (cf. section 2.3.2.b) et spécifie la dimension de chaque *block* de sorte que **Db.x * Db.y * Db.z** soit égal au nombre de *threads* par *block*
- **Ns** est de type **size_t** et spécifie le nombre d'octets dynamiquement alloués par *block* dans la mémoire partagée (voir l'exemple présenté section 2.3.1.b). Cet argument est optionnel, sa valeur par défaut est 0
- **S** est de type **cudaStream_t** et spécifie le *stream* (flot) associé à l'exécution (cf. section 2.3.4.c). Cet argument est optionnel, sa valeur par défaut est 0

Une fonction déclarée de cette façon...

```
1 __global__ void Premier_kernel(float* parametre);
```

... devrait donc être appelée de la sorte :

```
1 Premier_kernel<<< Dg, Db, Ns >>>(parametre);
```

Les arguments de la configuration seront, tout comme les arguments de la fonction, transmis via la mémoire partagée. Ils seront d'autre part traités avant même les arguments de la fonction.
L'exécution du *kernel* échouera si d'aventure **Dg** et/ou **Db** étaient plus grands que leur valeur maximale autorisée spécifiées en Annexe (cf. section B). De même le lancement du *kernel* ne pourrait s'effectuer si **Ns** était plus grand que la taille totale en octets pouvant être contenue dans la mémoire partagée, et à laquelle on aurait déjà retiré les tailles nécessaires au passage des arguments de la configuration

et de la fonction, ainsi que la taille dédiée à des variables déclarées à l'intérieur du *kernel* comme variables **___shared___**.

2.3.1.d Les variables standards

Les variables standards sont :
- **gridDim** : de type **dim3** (cf. section 2.3.2.b) elle contient la dimension de la grille
- **blockDim** : de type **dim3** (cf. section 2.3.2.b) elle contient la dimension d'un *block*
- **blockIdx** : de type **uint3** (cf. section 2.3.2.a) elle contient l'indice du *block* à l'intérieur de la grille
- **threadIdx** : de type **uint3** (cf. section 2.3.2.a), elle contient l'indice du *thread* à l'intérieur du *block*
- **warpSize** : de type **int**, elle contient la taille d'un *warp* en *threads*

Certaines restrictions sont à signaler :
- Il n'est pas possible d'obtenir l'adresse des variables standards
- Il n'est pas possible d'assigner une valeur à une variable standard

2.3.2 Des composants manipulables par le *host* et le *device*

2.3.2.a Des types vecteurs standards

Voici une liste, exhaustive à ce jour, de ces types :
char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2

Ces types sont des structures dont les composantes sont accessibles à travers les champs **x, y, z** et **w** respectivement. Ils sont fournis avec un constructeur de sorte qu'un vecteur de type **int2** et de composante (x', y') se déclare de la manière suivante :

```
1 int2 make_int2(int x, int y);
```

2.3.2.b Le type dim3

Ce type est fondé à partir du type **uint3**, et prévu pour des variables comportant des dimensions. Chaque champ non spécifié est initialisé à 1.

2.3.2.c Le type texture

Nous avons déjà eu l'occasion de présenter la *texturememory*, nous pouvons ajouter que son utilisation en lieu et place de la mémoire globale, peut améliorer les performances du programme de manière conséquente.

Les données présentes dans cette mémoire sont lues à partir d'un *kernel* en utilisant une fonction **___device___** appelée *texture fetch*. Le premier argument d'une telle fonction spécifie un objet appelé *texture reference*. Ces objets possèdent plusieurs attributs, dont l'un d'eux est sa dimension qui spécifie si la texture à laquelle on souhaite accéder est un vecteur de dimension 1, 2 ou 3. Les éléments de ce vecteur sont appelés *texels* (contraction de " *texture elements* ").

Une *texture reference* se déclare de la façon suivante :

```
1 texture<Type, Dim, ReadMode> texRef;
```

où **Type** précise le type de donnée qui sera retourné par une *texture fetch*, et ne peut prendre pour valeur que les type usuels et ceux définis section 2.3.4.a. D'autre part, **Dim**, qui est un argument optionnel à valeur par défaut égale à 1, précise la dimension de 'texRef' dans notre exemple. Enfin **ReadMode**, précise le mode de lecture comme son nom le suppose, et peut prendre une des deux valeurs suivantes : **cudaReadModeNormalizedFloat** ou **cudaReadModeElementType**, permettant respectivement soit de convertir le type de la texture en float normalisé (compris entre 0 et 1 pour des entiers non signés et entre -1 et 1 sinon), soit de ne pas faire de conversion.

Pour plus d'informations sur la *texturememory* et les fonctionnalités relatives à celle-ci, nous orientons le lecteur vers les tutoriaux directement téléchargeable depuis le site internet de NVIDIA (http ://www.nvidia.fr/page/home.html). En effet, ces notions n'ayant pas été utilisées par la suite lors de la programmation au moyen de CUDA, nous ne rentrerons pas d'avantage dans les détails.

2.3.3 Des composants manipulables par le *device*

2.3.3.a La fonction de synchronisation

Cette fonction a déjà été introduite dans ce chapitre (cf. section 2.1.2).

2.3.3.b Les fonctions atomiques

Nous avons invoqué l'utilité de ces fonctions atomiques, permettant d'assurer l'écriture à la même adresse par plusieurs *threads* simultanément, auparavant (cf.

section 2.2). Nous allons maintenant détailler leur fonctionnement.

Une telle fonction réalise les actions simultanées de lecture-modification-écriture suffisamment rapidement pour s'assurer que la variable ne soit pas altérée par une autre tentative d'écriture (c'est en ce sens que l'on dit que l'opération est atomique). Ces fonctions ne sont opérationnelles que sur des entiers signés et non signés (à l'exception de **atomicExch()** qui accepte aussi la manipulation de flottant simple précision). A titre d'exemple, la fonction **atomicAdd()** lit un mot de 32 bits se situant en mémoire globale ou partagée, y ajoute un entier, et écrit le résultat à la même adresse.

Toutes ces fonctions sont listées en Annexe (cf. section C). Comme précisé auparavant, toutes les cartes graphiques ne supportent pas ces fonctions et la *computecapability* minimale requise pour chacune de ces fonctions se trouve précisée en Annexe.

2.3.3.c Les fonctions *warps* de décisions

Ces fonctions ne s'adressent qu'aux cartes de *computecapability* de 1.2 et plus. Elles sont au nombre de deux :

```
1 int __all(int predicat);
```

évalue la variable 'predicat' pour tous les *threads* du *warp*, et retourne zéro si et seulement si, 'predicat' est évalué à zéro pour au moins un des *threads* du *warp*.

```
1 int __any(int predicat);
```

évalue la variable 'predicat' pour tous les *threads* du *warp*, et retourne zéro si et seulement si, 'predicat' est évalué à zéro pour tous les *threads* du *warp*.

2.3.4 Des composants manipulables par le *host*

2.3.4.a Les APIs

Le composant appelé *host* runtime se compose de deux Interfaces de Programmation (API) distinctes :
- Un premier niveau appelé *CUDA driver API*
- Un niveau supérieur appelé *CUDA runtime API*, implémenté par-dessus l'API précédente.

Ces APIs ne peuvent être utilisées conjointement par une application.

La *CUDA runtime API* est plus facile d'utilisation, car la *CUDA driver API* requiert d'avantage de code, et est plus difficile à débugger. En contrepartie cette dernière offre un meilleur niveau de contrôle mais aussi l'indépendance du langage informatique. En effet, elle ne manipule que des fichiers assemblage (code *ptx*) ou binaire (fichier *cubin*) qui sont générés par le compilateur nvcc introduit plus haut. Lors de la compilation, ce compilateur sépare le code pour différencier les parties programmées pour le *host* de celles programmées pour le *device*. Les parties programmées pour le *host* seront compilées à l'aide d'un autre compilateur, tandis que les parties restantes seront donc compilées sous forme de code *ptx* ou de fichier *cubin*.

La *CUDA driver API* est accessible à partir de la librairie dynamique *nvcuda*, et tous ces points d'entrée ont pour préfixe : **cu**.

La *CUDA runtime API* est accessible à partir de la librairie dynamique *cudart*, et tous ces points d'entrée ont pour préfixe : **cuda**.

Dans la suite nous ne traiterons que des fonctionnalités relatives à la *CUDA runtime API*. En effet, l'algorithme utilisé pour résoudre nos problèmes d'optimisation au moyen de CUDA se base sur cette API. Le choix s'est porté naturellement vers cette API pour sa prise en main plus évidente à partir du langage C. D'autre part la compréhension de l'algorithme par des personnes n'ayant jamais manipulé CUDA n'en est que plus aisée.

2.3.4.b Les instructions asynchrones exécutées en concurrence

Pour permettre l'utilisation du *host* et du *device* en concurrence, certaines fonctions sont, ou peuvent être, utilisées de façon asynchrone. Autrement dit, le contrôle est rendu au *host* avant même que le *device* n'ait terminé son exécution. Ces fonctions sont :
- Les *kernels*
- Les fonctions de copies de données du *host* vers le *device* ayant pour suffixe : **Async**
- Les fonctions de copies de données d'un *device* à un autre
- Les fonctions qui attribuent une valeur à une variable de la mémoire globale

Ces applications s'exécutent en concurrence sur plusieurs *streams* ou flots. Un tel *stream* est une séquence d'instruction se déroulant dans l'ordre, que l'on utilise comme argument de *kernels* ou de fonctions de copies mémoire entre le *host* et le *device*. Cette manipulation est décrite plus loin (cf. section 2.3.4.c).

A noter que les *kernels* lancés sans spécification concernant le *stream* à utiliser sont automatiquement assignés au *stream* 0. Il en est de même pour les fonctions de recopies non suivies du suffixe **Async**.

La fonction **cudaStreamSynchronize()** permet de forcer l'attente de la fin de toutes les instructions s'effectuant sur le *stream* en question. Cette fonction possède d'ailleurs son équivalent concernant l'ensemble des *threads* d'une grille : **cudaThreadsSynchronize()**. Il est tout de même recommandé de ne pas abuser de ces fonctions, qui sont surtout utiles pour déterminer le temps d'exécution d'une instruction particulière.

Il est possible, notamment pour debugger un programme, d'imposer la valeur 1 à la variable d'environnement **CUDA_LAUNCH_BLOCKING** afin d'empêcher tout asynchronisme à l'intérieur d'une application.

2.3.4.c *RuntimeAPI*

Contrairement à la *CUDA driver API*, il n'est pas nécessaire de procéder à une initialisation pour la *runtime API*. Celle-ci sera effectuée automatiquement dès le premier appel à une fonction de la *runtime API*.

Certaines de ces fonctions permettent d'avoir accès aux caractéristiques des cartes graphiques installées sur l'ordinateur. L'exemple suivant permet de récupérer l'ensemble de ces caractéristiques pour toutes les cartes installées :

```
1 int deviceCount;
2 cudaGetDeviceCount(&deviceCount);
3 int device;
4 for (device = 0; device < deviceCount; ++device)
5 {
6 cudaDeviceProp deviceProp;
7 cudaGetDeviceProperties(&deviceProp, device);
8 }
```

La fonction **cudaSetDevice()** est quant à elle utilisée pour choisir le *device* à relier avec le *host*. Si aucun appel à cette fonction n'est employé avant le premier lancement d'un *kernel*, alors le *device* 0 sera sélectionné automatiquement et il ne sera plus possible de faire appel à **cudaSetDevice()** dans la suite du programme.

En ce qui concerne la mémoire du *device*, elle se manipule à l'aide de fonctions telle que **cudaMalloc()**, pour l'allocation mémoire, **cudaFree()**, pour libérer la mémoire, ou encore **cudaMemcpy()** pour copier la mémoire. Les deux exemples suivants permettent respectivement de copier de la mémoire depuis le *host* vers la mémoire globale, ou vers la mémoire constante :

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Certaines fonctions permettent également d'allouer de la mémoire spécialement pour les vecteurs de types standards définis section 2.3.2.a.

Enfin pour ce qui est des *streams*, l'exemple suivant permet tout d'abord de créer deux *streams*. Ensuite chaque *stream* permet la copie d'une portion du vecteur 'hostPtr' résidant sur le *host* vers le vecteur 'inputDevPtr' se trouvant, lui, sur le *device*. Puis chacun d'eux effectue quelques opérations sur ce dernier vecteur dont les modifications sont apportées au vecteur 'outputDevPtr' (lui aussi sur le *device*), avant de recopier sur le *host*, dans le vecteur d'origine, les données du vecteur 'outputDevPtr'.

```
cudaStream_t stream[2];

for (int i = 0; i < 2; ++i)
cudaStreamCreate(&stream[i]);
for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
    cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 2; ++i)
myKernel<<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i
    * size, size);

for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
    cudaMemcpyDeviceToHost, stream[i]);

cudaThreadSynchronize();
```

2.3.4.d Le mode émulation

CUDA n'est pas muni d'un debugger à proprement parler. Le mode émulation permet cependant de résoudre certains problèmes.
En effet, en utilisant l'option de compilation **-deviceemu**, le compilateur est forcé de compiler l'ensemble de l'application pour le *host*. Par conséquent, tout le programme sera ainsi exécuté sur CPU. Bien entendu il y a certains paramètres à vérifier avant de lancer le mode émulation :
- Le *host* doit être capable de faire tourner jusqu'au nombre maximum de *threads*

par *block* plus un pour le maître
- Le *host* doit posséder suffisamment de ressources pour chaque *threads*, c'est-à-dire 256 KB

L'utilisation d'un tel mode permet donc d'utiliser les moyens habituels pour debugger un programme. De plus, toutes les variables résident alors sur le *host*. De ce fait, il est possible d'abuser d'affichages écran pour mieux comprendre les erreurs de codage.

Cependant, le fait de constater que son programme fonctionne en mode émulation ne garantie pas du tout qu'il en soit de même sur GPU. La valeur des variables prévues pour le *device* peut sembler cohérente en mode émulation mais ne pas l'être lors de l'exécution sur carte graphique. Un message d'erreur typique déclenché par une exécution erronée sur GPU est **unspecifief launch failure** (soit l'équivalent du 'segmentation fault' du langage C). La cause de ce problème peut ne pas se déclencher lorsque toutes les données (tous les pointeurs) résident sur le *host*. D'autre part, il est tout à fait possible d'observer des différences de précisions au niveau des résultats. Enfin la taille d'un *warp* est alors de 1 *thread*, par conséquent l'appel à une fonction *warp* de décisions peut donner un résultat inattendu.

2.4 Règle d'optimisation

2.4.1 Instructions basiques

La plupart des simples instructions ne nécessitent que quatre cycles d'horloge. C'est le cas par exemple des instructions suivantes :
- Addition d'entiers
- Opération bit-à-bit
- Comparaison
- Conversion de type

En revanche il est recommandé d'éviter les divisions autant que possible. En particulier lors de l'instruction i/n si n est une puissance de 2, alors il convient de remplacer l'instruction précédente par une équivalente : $i >> log2(n)$. De la même façon $i\%n$ (reste de la division euclidienne de i par n) sera plutôt implémenté de la façon suivante : $i\&(n-1)$. Si n est littéral la conversion est automatique.

2.4.2 Instructions de contrôle

Les instructions de contrôle telles que **if, switch, do, for,** ou encore **while** peuvent considérablement ralentir un *kernel*. En effet, pas essence elles créent des chemins différents à l'intérieur d'une même application, autrement dit des divergences au sein d'un *warp*. Chacun de ces chemins créés sera alors exécuté en série jusqu'à ce qu'ils convergent dans la même direction, et que le parallélisme s'applique à nouveau, comme précisé précédemment (cf. section 2.2).

Malheureusement ces instructions sont parfois inévitables. Dans de tels cas, il est conseillé, afin d'éviter les divergences de *threads*, de ne pas appliquer la condition sur l'identifiant d'un *thread* mais de faire en sorte qu'elles dépendent plutôt du *warp*. En d'autres termes, il est possible de minimiser les divergences en alignant le conditionnement sur les *warps*, les instructions porteuses de divergences dépendront donc du résultat de **threadIdx/WSIZE** où **WSIZE** est la taille d'un *warp*.

2.4.3 Instructions de gestion mémoire

Un multiprocesseur requiert 4 cycles d'horloge pour exécuter une instruction de gestion de la mémoire (lecture, écriture) pour un *warp*. Cependant lorsqu'il faut accéder aux mémoires globale ou locale, il faut prévoir entre 400 et 600 cycles d'horloge relatifs au temps de latence. Ce temps de latence peut cependant être dissimulé en codant de sorte que le *kernel* possède un ratio important d'instructions de calculs par rapport aux transactions mémoire. Malgré ces quelques généralités, les différentes mémoires possèdent des caractéristiques très diverses qu'il est bon de maitriser pour éviter une perte de performance non négligeable.

Dans la suite nous ne considérons que les cas où la carte graphique possède une *computecapability* de 1.2 minimum.

2.4.3.a Mémoire globale

La mémoire globale ne possède aucun cache, il est donc très important de suivre le mode d'accès aux données prévu pour maximiser la bande passante.

Ce mode d'accès est conçu pour permettre à l'ensemble des *threads* d'un $half - warp$ (la moitié d'un *warp*) d'atteindre la mémoire globale en une seule transaction mémoire (on dira que les accès mémoires sont amalgamés). Pour ce faire, ces accès mémoire doivent respecter certaines règles.

Ainsi il est notamment nécessaire que tous les *threads* d'un $half - warp$ accèdent à des données résidant dans un segment (zone d'éléments mémoire contigus) de taille :
- 32 octets si ces *threads* accèdent à des mots de 8 bits,
- 64 octets si ces *threads* accèdent à des mots de 16 bits,
- 128 octets si ces *threads* accèdent à des mots de 32 ou 64 bits

Si en revanche, le $half - warp$ accède à des mots dans n segments différents (accès non contigus), n transactions mémoires seront réalisées.

A noter que les éléments d'un segment auquel aucun *thread* du $half - warp$ n'accède sont tout de même lus.

Les figures en Annexe (section 7.2.4.a) permettent de mettre en évidence les situations d'accès amalgamés et non amalgamés.

2.4.3.b Mémoire locale

L'accès à la mémoire locale n'intervient que lorsque des variables automatiques sont affectées non pas aux registres par manque de ressources, mais à la mémoire dite locale. Tout comme la mémoire globale, la mémoire locale ne possède pas de cache, les accès sont donc aussi coûteux que pour la mémoire globale. Les accès sont cependant toujours amalgamés puisque par définition, pour la mémoire locale, les accès se font par *thread* et non par *warp*.

2.4.3.c Mémoire constante

A la différence des deux mémoires précédentes, la mémoire constante possèdent un cache. Par conséquent, pour tous les *threads* d'un $half - warp$, lire des données situées dans le cache de la mémoire constante est aussi rapide que pour des données situées dans des registres tant que tous ces *threads* lisent à la même adresse. Les coûts d'accès augmentent linéairement avec le nombre d'adresses distinctes accédées par les *threads*.

2.4.3.d Mémoire partagée

La mémoire partagée est définitivement plus rapide que les mémoires globale ou locale. En fait, pour chaque *thread* d'un *warp*, accéder à la mémoire partagée est tout aussi rapide que d'accéder aux registres sous réserve cependant que ces accès

mémoires se fassent sans conflit. Nous détaillerons ces conflits plus loin et les appellerons désormais conflits de *banks*.

Pour offrir des transferts de données efficaces, la mémoire partagée est divisée en segments de tailles égales appelés *banks*, auxquels il est possible d'accéder simultanément. Cependant, si deux requêtes tombent dans la même bank, il y a alors création d'un conflit de bank, et les accès seront réalisés en série. Il est donc important de comprendre comment manipuler la mémoire partagée pour minimiser ces conflits.

Les *banks* sont organisées de sorte que des mots successifs de 32 bits chacun soient assignés à des *banks* successives, et chaque *bank* possède une bande passante de 32 bits pour 2 cycles d'horloge. Pour les cartes de *computecapability* 1.x, un *warp* est de taille 32, et le nombre de *bank* est de 16. De ce fait, les requêtes sont partagées entre le premier $half - warp$ et le second. Par conséquent, il ne peut y avoir de conflit de *bank* entre deux *threads* résidant dans deux moitiés différentes d'un même *warp*.

Généralement, l'accès à la mémoire partagée se fait de sorte que chaque *thread* ait besoin d'atteindre un mot de 32 bits d'un vecteur indexé via l'identifiant du *thread* (que nous noterons tid) et un saut noté s :

```
1 __shared__ float shared[32];
2 float data = shared[BaseIndex + s * tid];
```

Dans un cas comme celui-ci, le *thread* **tid** et le *thread* **tid+n** accèdent à la même *bank* chaque fois que **s*n** est un multiple du nombre de *banks* **m**, soit encore, chaque fois que **n** est un multiple de **m/d** où **d** est le plus grand diviseur commun de **m** et **s**. Par conséquent il n'y aura pas de conflit si et seulement si la taille du $half - warp$ est inférieure ou égale à **m/d**. Autrement dit, concernant les cartes graphiques de *computecapability* de 1.x, il ne reste pas d'autre possibilité que **d=1**, i.e. s doit être impair puisque **m** est une puissance de 2. Les deux premières figures présentées en Annexe (section 7.2.4.b) montrent quelques exemples d'accès sans conflits, en revanche la figure qui suit (Annexe section 7.2.4.b) dévoile des exemples de conflits.

Enfin, il est possible, lors d'une requête visant à atteindre la même adresse pour tous les *threads* d'un $half - warp$, que cette adresse soit " broadcastée " (distribuée à chaque *thread* sans aucun conflit de bank). Ceci ne se produit évidemment que dans le cas où le mot accédé est de taille 32 bits. Ce mécanisme se met également en place, lorsque seulement quelques *threads* du $half - warp$ accèdent à la même adresse. En effet, dans de tels cas, le mot est également " broadcasté " (si plusieurs mêmes mots sont accédés simultanément alors il n'y a aucune garantie sur le choix du mot " broadcasté "), réduisant ainsi les conflits. La dernière figure en Annexe (section 7.2.4.b) expose quelques exemples de broadcasts.

2.4.3.e Nombre de *threads* par *block*

Une première règle à prendre en considération lors du choix de la configuration, est de prévoir un nombre de *blocks* au moins égal au nombre de multiprocesseurs. D'autre part, n'utiliser qu'un seul *block* par multiprocesseur, obligerait celui-ci à rester inactif durant les synchronisations éventuelles des *threads*, et le temps de latence lors des accès aux mémoires globale et locale ne sera pas caché si le nombre de *threads* par *block* n'est pas suffisamment important.

Il est également important de choisir un nombre de *threads* par *block* en tant que multiple de la taille d'un *warp* pour éviter un gaspillage avec des *warps* sous-peuplés. Cependant plus le nombre de *threads* par *block* est important, moins il y aura de registres disponibles. Cet inconvénient pourrait donc conduire à une utilisation trop importante de la mémoire locale, peu efficace en terme de temps d'accès.

De façon concrète lorsqu'il s'agit de déterminer la configuration optimale de la grille pour un *kernel* particulier, il est possible de compiler son application via l'option de compilation **-ptxas-options=-v**. Cette option commande au compilateur **nvcc** la création d'un fichier *cubin* à l'intérieur duquel il est possible de lire les informations concernant l'espace mémoire requit par le *kernel* :

```
1  architecture {sm_10}
2  abiversion {0}
3  modname {cubin}
4  code {
5          name = BlackScholesGPU
6          lmem = 0
7          smem = 68
8          reg = 20
9          bincode {
10                 0xa0004205 0x04200780 0x40024c09 0x00200780
```

L'exemple ci-dessus permet donc de rendre compte de l'utilisation de 20 registres par *threads*, mais également 68 octets de mémoire partagée par *block*. A partir de ces données, il est donc possible de déterminer une configuration optimale pour la grille d'un tel *kernel*. Ensuite, ces données peuvent être entrées dans le calculateur (fichier excel) disponible sur le site de NVIDIA, pour obtenir le nombre de *threads* par *block* susceptible d'offrir une occupation maximale pour un multiprocesseur donné.

2.4.3.f Transferts de données CPU <-> GPU

Il est recommandé de faire le moins de transfert de données possible entre les deux entités. Il est également préférable à priori, de transvaser davantage de code

sur carte graphique quitte à n'utiliser qu'un faible parallélisme. Enfin un transfert important de données est plus intéressant que plusieurs transferts moindres, du fait du temps de latence.

3 Acquisition de l'image

Modélisation de l'acquisition d'images par satellite

Soit u une image que l'on souhaite restaurer, avec $u : \mathbb{R}^2 \to \mathbb{R}$ (pour des images en noir et blanc), $u : \mathbb{R}^2 \to \mathbb{R}^3$ (pour des images couleurs).
Lors de l'acquisition de l'image, la lentille crée un phénomène de diffraction que nous pouvons modéliser de la façon suivante :

$$u' = Au$$

où u' est l'image de la scène u observée à travers la lentille.

Par la suite, l'image u' est acquise au moyen d'un capteur CCD placé dans le plan focal de la lentille. Notons u_0 l'image obtenue après discrétisation :

$$u_0 = Su'$$

soit encore

$$u_0 = SAu + b$$

où b est un bruit photonique du capteur que l'on suppose gaussien.

Le but du projet est de résoudre le problème inverse : retrouver l'image originale u à partir de u_0.

4 Débruitage

Nous nous intéressons dans un premier temps au problème du débruitage. Pour ce faire, nous cherchons à retrouver (ou tout du moins approcher) l'image telle qu'elle était avant d'être perturbée par le bruit photonique. Cela se traduit par la recherche de l'image u^* qui maximisera la probabilité que cette image soit celle correspondant à la scène observée originale u. Ce principe est connu sous le nom de **Maximum a posteriori**. On peut le formuler mathématiquement par :

$$\max_{u \in \mathbb{R}^n} P(u|u_0) \tag{1}$$

En utilisant la formule de Bayes :

$$P(u|u_0) = \frac{P(u_0|u).P(u)}{P(u_0)},$$

nous pouvons alors reformuler le problème de la manière suivante :

$$\max_{u\in\mathbb{R}^n} \frac{P(u_0|u).P(u)}{P(u_0)}$$

$$\max_{u\in\mathbb{R}^n} P(u_0|u).P(u)$$

Puis,

$$\min_u -ln(P(u_0/u)) - ln(P(u))$$

Comme $u_0 = u + b$ et que le bruit est considéré comme gaussien et indépendant de u, on peut écrire :

$$P(u_0|u) = P(u + b|u) = P(b) = e^{-\frac{\|u_0-u\|_2^2}{2\sigma^2}}$$

avec σ^2 la variance du bruit. Il reste à déterminer $P(u)$. Supposons que cette probabilité s'écrive sous la forme :

$$P(u) = e^{-J(u)}$$

où J est une fonction positive. Le problème d'optimisation (1) s'écrit alors :

$$\min_u \frac{\|u_0 - u\|_2^2}{2\sigma^2} + J(u) \Leftrightarrow (1)$$

Trouver une fonctionnelle $J(u)$ qui modélise bien l'espace des images est un problème compliqué. Dans ce travail on admettra qu'une image est d'autant plus probable qu'elle est régulière. Une façon simple de l'écrire consiste à choisir par exemple :

$$J(u) = \int_\Omega |\nabla u|^2 d\Omega$$

Dans ce cas,u est modélisé comme un élèment de l'espace de Sobolev, $H_1(\Omega) = \{u \in L^2(\Omega) : \nabla u \in L^2(\Omega)\}$. L'inconvénient d'un tel espace est qu'il exige de ses éléments une régularité trop importante pour être réaliste en traitement d'images. Plus une image est régulière, moins on distingue les contours des objets.

Par conséquent, on doit utiliser un espace fonctionnel plus grand qui contient des fonctions à sauts. Un espace communément utilisé est l'espace $BV(\Omega)$, l'espace des fonctions à variations bornées défini par

$$BV(\Omega) = \{u \in L^1(\Omega), sup(\int_\Omega u\ div\varphi\ d\Omega) < +\infty \text{ où } \varphi \in C_0^\infty(\Omega),\ \text{et } \|\varphi\|_\infty \leq 1\}$$

Cet espace comporte des fonctions moins régulières, mieux adaptées aux problèmes liés à l'imagerie. L'opérateur gradient n'étant cependant pas défini dans cet espace,

il est plus compliqué à décrire.
Le problème (1) s'écrit dans cet espace sous la forme :

$$\min_{u \in BV(\Omega)} \frac{\|u_0 - u\|_2^2}{\sigma^2} + TV(u)$$

où TV (acronyme anglais pour Total Variation) est défini par :

$$TV(u) = sup(\int_\Omega u \; div\varphi \; d\Omega), \text{ avec } \varphi \in C_0^\infty(\Omega) \text{ et } \|\varphi\| \leq 1.$$

Dans le cas où u appartient à l'espace de Sobolev $W^{1,1}(\Omega) = \{u \in L^1(\Omega) | \nabla u \in (L^1(\Omega))^2\}$, $TV(u) = \int_\Omega |\nabla u| \; d\Omega$ où ∇u doit être compris au sens des distributions.
Après discrétisation, on obtient :

$$\min_{u \in \mathbb{R}^n} \frac{\sum_{i=0}^n |u_0(i) - u(i)|^2}{2\sigma^2} + \sum_{i=0}^n \sqrt{(\partial_1 u)^2(i) + (\partial_2 u)^2(i)}$$

On introduit alors l'application :

$$|||.|||_1 : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$$
$$z = \begin{pmatrix} x \\ y \end{pmatrix} \to \sum_{k=0}^n \sqrt{x^2(k) + y^2(k)}$$

qui est une norme sur $\mathbb{R}^{2n}$.
On pose $\Psi = \frac{\|u_0 - u\|_2^2}{\sigma^2} + \sum_{i=0}^n \sqrt{(\partial_1 u)^2(i) + (\partial_2 u)^2(i)}$.
La norme étant convexe, la fonction que nous cherchons à minimiser est donc convexe. Cela nous assure de l'existence d'un minimum global. Le problème de minimisation (1) s'écrit ainsi :

$$\min_{u \in \mathbb{R}^n} \frac{\|u_0 - u\|_2^2}{2\sigma^2} + |||\nabla u|||_1$$

4.1 Résolution du problème par l'algorithme de descente de gradient

Dans un premier temps, nous allons résoudre le problème avec un algorithme de descente de gradient. Cet algorithme utilise le gradient de l'itération précédente pour calculer la nouvelle valeur de la fonction à minimiser. Une itération s'écrit :

$$x^{k+1} = x^k - \tau \times \nabla f(x).$$

Le problème consiste à trouver un pas τ qui assure la convergence de l'algorithme. Le pas doit satisfaire la contrainte suivante :

$$\tau \leq \frac{2}{L}$$

où L est la constante de Lipschitz de $\nabla f(x)$. Dans les deux paragraphes suivants, nous calculerons le gradient et le pas.

4.1.1 Calcul du gradient

La fonction peut être divisée en deux parties : $\Psi_1 = \frac{\|u_0-u\|_2^2}{2\sigma^2}$ et $\Psi_2 = \||\nabla u|\|_1$.
On a

$$\nabla\Psi_1(u) = \frac{u-u_0}{\sigma^2}$$

Calculons maintenant $\nabla\Psi_2(u)$:

On écrit $\||\nabla u|\|_1$ sous la forme $\varphi(Au)$ où :

$$\varphi : \mathbb{R}^{2n} \to \mathbb{R}^n$$

$$\varphi : \begin{pmatrix} x \\ y \end{pmatrix} \to \sum_k \sqrt{x^2(k) + y^2(k) + \epsilon^2}$$

On pose $A = \nabla$

$$\varphi(A(u+h)) = \varphi(Au + Ah) = \varphi(Au) + \nabla\varphi(Au)(Ah) + o(\|h\|_2)$$

On a : $\nabla\varphi : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ et$\nabla\varphi = \begin{pmatrix} \frac{x(k)}{\sqrt{x^2(k)+y^2(k)+\epsilon^2}} \\ \frac{y(k)}{\sqrt{x^2(k)+y^2(k)+\epsilon^2}} \end{pmatrix}$, ce qui donne :

$$(\nabla\varphi(\nabla u))_{i,j} = (-div(\frac{\nabla(u)}{\sqrt{(\partial_1 u)^2 + (\partial_2 u)^2 + \epsilon^2}}))_{i,j}$$

où la division doit être comprise comme une division terme à terme.
On a donc :

$$\nabla\Psi(u)_{i,j} = \frac{u_{i,j} - u_{0,ij}}{\sigma^2} - (div(\frac{\nabla u}{\sqrt{(\partial_1 u)^2 + (\partial_2 u)^2 + \epsilon^2}}))_{i,j}$$

Il faut donc évaluer maintenant $\partial_1 u$ et $\partial_2 u$. Pour cela on utilise les notations suivantes (selon Chambolle) :

$$(\nabla u)_{i,j} = ((\partial u)^1_{i,j}, (\partial u)^2_{i,j})$$

Avec :

$$(\nabla u)^1_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j} \text{ si } i < N \\ 0 \text{ sinon} \end{cases}$$

$$(\nabla u)^2_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j} \text{ si } j < N \\ 0 \text{ sinon} \end{cases}$$

Et

$$(div\ p)_{i,j} = \begin{cases} p^1_{i,j} - p^1_{i-1,j} \text{ si } 1 < i < N \\ p^1_{i,j} \text{ si } i = 1 \\ -p^1_{i,j} \text{ si } i = N \end{cases} + \begin{cases} p^2_{i,j} - p^2_{i,j-1} \text{ si } 1 < j < N \\ p^2_{i,j} \text{ si } j = 1 \\ -p^2_{i,j} \text{ si } j = N \end{cases}$$

On pose :

$$N_{i,j} = \sqrt{(u_{i+1,j} - u_{i,j})^2 + (u_{i,j+1} - u_{i,j})^2 + \epsilon^2}$$

Les formules compactes sont utilisables en Matlab, mais pas en CUDA, pour cela nous allons développer plus les calculs. On obtient :
Si $1 < i < N$ et $1 < j < N$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} + u_{i,j+1} - 2u_{i,j}}{N_{i,j}} - \frac{u_{i,j} - u_{i-1,j}}{N_{i-1,j}} - \frac{u_{i,j} - u_{i,j-1}}{N_{i,j-1}}$$

Si $1 < i < N$ et $j = 1$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i,j+1}}{N_{i,j}} - \frac{u_{i,j} - u_{i-1,j}}{N_{i-1,j}}$$

Si $1 < i < N$ et $j = N$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\sqrt{(u_{i+1,j} - u_{i,j})^2 + \epsilon^2}} - \frac{u_{i,j} - u_{i-1,j}}{\sqrt{(u_{i,j} - u_{i-1,j})^2 + \epsilon^2}}$$

Si $i = 1$ et $1 < j < N$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{N_{i,j}} + \frac{u_{i,j+1} - u_{i,j}}{N_{i,j}} - \frac{u_{i,j} - u_{i,j-1}}{N_{i,j-1}}$$

Si $i = N$ et $1 < j < N$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\sqrt{(u_{i,j+1} - u_{i,j})^2 + \epsilon^2}} - \frac{u_{i,j} - u_{i,j-1}}{\sqrt{(u_{i,j} - u_{i,j-1})^2 + \epsilon^2}}$$

Si $i = 1$ et $j = 1$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{N_{i,j}} + \frac{u_{i,j+1} - u_{i,j}}{N_{i,j}}$$

Si $i = N$ et $j = N$:

$$\nabla\Psi(u)_{i,j} = 0$$

Si $i = 1$ et $j = N$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\sqrt{(u_{i+1,j} - u_{i,j})^2 + \epsilon^2}}$$

Si $i = N$ et $j = 1$:

$$\nabla\Psi(u)_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\sqrt{(u_{i,j+1} - u_{i,j})^2 + \epsilon^2}}$$

4.1.2 Calcul du pas

On rappelle le résultat de convergence suivant : l'algorithme de descente de gradient converge vers la solution optimale si le pas d'itération est inférieur à : $\frac{2}{\lambda}$, où λ est la constante de Lipschitz de $\nabla\Psi(u)$ que nous allons calculer.

On a :

$$\|\nabla\Psi(u) - \nabla\Psi(v)\| = \|\frac{u-v}{\sigma^2} + \nabla TV_\epsilon(u) - \nabla TV_\epsilon(V)\|$$

avec $\nabla TV_\epsilon(u) = \nabla^T(\frac{\nabla(u)}{\sqrt{\epsilon^2+|\nabla u|_2^2}})$. C'est à dire $\nabla TV_\epsilon(u) = \varphi(\nabla u)$
On a donc :

$$\|\nabla\Psi(u) - \nabla\Psi(v)\| \leq \frac{\|u-v\|}{\sigma^2} + |||\nabla^T|||\ \|\varphi(\nabla u) - \varphi(\nabla v)\|$$

Calculons la dérivée de φ : soit $x \in \mathbb{R}$,

$$\forall x \in \mathbb{R}^{2n},\ \varphi_\epsilon^{'} = \frac{\sqrt{\epsilon^2+|x|^2} - x \times \frac{x}{\sqrt{\epsilon^2+|x|^2}}}{\epsilon^2+|x|^2} = \frac{\epsilon^2}{(\epsilon^2+|x|^2)^{\frac{3}{2}}} \leq \frac{1}{\epsilon}$$

D'après le théorème des accroissements finis on peut donc écrire :

$$\|\varphi_\epsilon^{'}(\nabla u) - \varphi_\epsilon{}'(\nabla v)\| \leq \frac{1}{\epsilon}\|\nabla u - \nabla v\| \leq \frac{|||\nabla|||}{\epsilon}\|u-v\|.$$

Et donc il vient :

$$\|\nabla\Psi(u) - \nabla\Psi(v)\| \leq \frac{\|u-v\|}{\sigma^2} + \frac{|||\nabla|||^2}{\epsilon}\|u-v\|.$$

Et donc :

$$\|\nabla\Psi(u) - \nabla\Psi(v)\| \leq (\frac{|||\nabla|||^2}{\epsilon} + \frac{1}{\sigma^2})\|u-v\|.$$

On sait d'après Chambolle que $|||\nabla|||^2 = 8$. Le pas est donc donné par

$$\tau = \frac{2}{\frac{8}{\epsilon} + \frac{1}{\sigma^2}}$$

4.2 Résolution du problème par une approche duale

Une autre méthode consiste à passer par le problème dual que l'on résoudra par un algorithme de gradient projeté. L'intérêt de cette nouvelle appproche est d'éviter la régularisation qui fait exploser les constantes de Lipschitz. Théoriquement cette approche devrait permettre une convergence plus rapide
Rappel du problème à résoudre :

$$\min_{u\in\mathbb{R}^n} \frac{\|u_0-u\|_2^2}{2\sigma^2} + |||\nabla u|||_1 \tag{2}$$

Afin de déterminer la formulation du problème dual nous cherchons $|||q|||_\infty$ de sorte que l'on puisse écrire :

$$\max_{q\in\mathbb{R}^{2n},|||q|||_\infty\leq 1} <\nabla u, q> = |||\nabla u|||_1$$

La norme l-infinie que nous introduisons sur $\mathbb{R}^{2n}$ est définie par :

$$|||q|||_\infty = \max_{i\in[1,..,n]} \sqrt{(q_{1i})^2+(q_{2i})^2}$$

Enfin, le problème (2) peut se reformuler comme suit :

$$\min_{U\in\mathbb{R}^n} \max_{q\in\mathbb{R}^{2n},|||q|||_\infty\leq 1} \frac{\|u_0-u\|_2^2}{2\sigma^2} + <\nabla u, q>$$

Le théorème de dualité de Fenchel nous permet alors d'écrire :

$$\max_{q\in\mathbb{R}^{2n},|||q|||_\infty\leq 1} \min_{u\in\mathbb{R}^n} \frac{\|u_0-u\|_2^2}{2\sigma^2} + <\nabla u, q> \tag{3}$$

Posons Ψ la fonction qui à $u\in\mathbb{R}^n$ associe $\frac{\|u_0-u\|_2^2}{2\sigma^2} + <\nabla u, q>$.
Résoudre (3) revient à chercher les solutions de $\nabla\Psi(u)=0$ soit les u tels que :

$$\frac{u-u_0}{\sigma^2} + \nabla^T q = 0$$

soit

$$u = u_0 - \sigma^2\nabla^T q \tag{4}$$

En réinjectant la solution trouvée ci-dessus dans le problème (3 on obtient pour le problème interne :

$$\begin{aligned}
\Psi(u_0-\sigma^2\nabla^T q) &= \frac{\|u_0-u_0+\sigma^2\nabla^T q\|}{2\sigma^2} + <u_0-\sigma^2\nabla^T q, \nabla^T q> \\
&= \frac{\sigma^2}{2}\|\nabla^T q\|_2^2 + <u_0-\sigma^2\nabla^T q, \nabla^T q> \\
&= \frac{-\sigma^2}{2}\|\nabla^T q\|_2^2 + <u_0, \nabla^T q>
\end{aligned}$$

Le problème dual s'écrit donc :

$$\max_{q\in\mathbb{R}^{2n},|||q|||_\infty\leq 1} < u_0, \nabla^T q > -\frac{\sigma^2}{2}\|\nabla^T q\|_2^2$$

soit encore :

$$\min_{q\in\mathbb{R}^{2n},|||q|||_\infty\leq 1} \frac{\sigma^2}{2}\|\nabla^T q\|_2^2 - < \nabla u_0, q >$$

On définit $\Theta(q) = \frac{\sigma^2}{2}\|\nabla^T q\|_2^2 - < \nabla u_0, q >$.
Nous pouvons maintenant écrire l'algorithme du gradient projeté pour résoudre ce problème. Voyons ce que donne une itération :

$$q^{k+1} = \Pi_K(q^k - \tau^k \nabla\Theta(q^k))$$

où $\tau^k = \frac{1}{4\sigma^2}$ et $\nabla\Theta(q^k) = -\sigma^2\nabla(div(q^k)) - \nabla u_0$, et $K = \{q \in \mathbb{R}^{2n} | |||q|||_1 \leq 1\}$.

La suite des itérés construits par ce procédé converge vers $q^* \in \mathbb{R}^{2n}$ et nous donne : $u = u_0 - \sigma^2\nabla^T q^*$ d'après (4). Pour cela nous avons donc besoin de définir la projection Π_K et les $\nabla\Theta(q^k)$.
On remarque que la variation totale qui tendait à trop régulariser dans le problème primal est absente ici.

4.2.1 Détermination du gradient

On rappelle que $\Theta(q) = \frac{\sigma^2}{2}\|\nabla^T q\|_2^2 - < \nabla u_0, q >$.
On a :

$$\nabla\left(\frac{\sigma^2}{2}\|\nabla^T q\|_2^2\right) = \sigma^2\nabla(\nabla^T q) = -\sigma^2\nabla(div(q))$$

d'où $\nabla\Theta(q) = -\sigma^2\nabla(div(q)) - \nabla u_0$.

4.2.2 Determination de la constante de Lipschitz du Gradient

Calculons :

$$\begin{aligned}
\|\nabla\Theta(q) - \nabla\Theta(p)\| &= \|\sigma^2\nabla(div(q-p)) - \nabla u_0 + \nabla u_0\| \\
&= \sigma^2\|\nabla(div(q-p))\| \\
&\leq \sigma^2|||\nabla|||\|div(q-p)\| \\
&\leq \sigma^2|||\nabla|||^2\|q-p\|
\end{aligned}$$

La constante de Lipschitz du gradient est donc $\sigma^2|||\nabla|||$ avec $|||\nabla|||^2 = 8$.

Le pas de l'algorithme étant donné par la formule $\tau = \frac{2}{\lambda}$ où λ est la constante de Lipschitz du gradient, elle vaut ici :

$$\lambda = 8\sigma^2 \text{ soit } \tau \leq \frac{1}{4\sigma^2}$$

4.2.3 Projection sur l'ensemble K

On introduit l'application :

$$\Pi_K(y) = \begin{cases} y \text{ si } |y| \leq 1 \\ \frac{y}{|y|} \text{ sinon} \end{cases}$$

Cette projection vise à ramener le vecteur y dans l'ensemble où l'on cherche nos solutions.

4.3 Résultats

Nous allons maintenant exposer les résultats obtenus par les deux méthodes, ainsi que les temps de calculs obtenus avec Matlab et ceux de CUDA. Voici l'image originale (non floutée et bruitée) qui servira de base pour nos tests. Sa taille est de 512×512 pixels.

FIG. 4 – Image originale

En utilisant les même paramètres, on obtient naturellement les mêmes résultats avec Matlab et CUDA. Le critère d'arrêt est défini comme étant le nombre d'itrations affiché en légende. Pour initialiser l'algorithme on prend l'image bruitée, le choix des paramètres ϵ et σ est arbitraire et fondé sur les expériences réalisées.

4.3.1 Descente de gradient


FIG. 5 – Image bruitée avec un bruit gaussien de $\sigma^2 = 0.1$, 50 itérations, 9.5 s avec Matlab et 82 ms avec CUDA dont 38.6 de copie en mémoire


FIG. 6 – Image bruitée avec un bruit gaussien de $\sigma^2 = 0.1$, 100 itérations, 18.4 s avec Matlab et 138 ms avec CUDA dont 74.1 ms de copie en mémoire

FIG. 7 – Image bruitée avec un bruit gaussien de $\sigma^2 = 0.2$, 100 itérations, 19.5 s avec Matlab et 83 ms avec CUDA dont 39 ms de copie en mémoire

4.3.2 Gradient projeté

FIG. 8 – Image bruitée avec un bruit gaussien de $\sigma^2 = 0.1$, 50 itérations, 4.1 s

FIG. 9 – Image bruitée avec un bruit gaussien de $\sigma^2 = 0.2$, 50 itérations, 4.3 secondes

On voit ici qu'en 50 itérations l'algorithme a l'air mieux convergé qu'avec une descente de gradient ordinaire, et les résultats sont très bons. Il faut 4.1 secondes sous Matlab pour obtenir le dernier résultat. Avec CUDA, on a un gain de calcul qui monte jusqu'à 130 : un débruitage d'image est en moyenne 120 fois plus rapide, le temps de calcul en lui-même est très réduit, la copie en mémoire prenant beaucoup de temps. Ces résultats ont été obtenus pour une image de taille 512×512, et pour des images plus grandes le gain en calcul est aussi plus important pourvu que chaque pixel de l'image puisse être traitée par un thread de CUDA.

5 Déconvolution

Dans ce chapitre nous allons exposer la méthode de résolution par déconvolution. Lorsque que l'on prend une photo, elle peut être brouillée (ou floutée) pour différentes raisons : à cause d'un mouvement de l'objet observé, d'une aberration optique sur la lentille (ou le miroir) ou même dans le cas de l'astronomie à cause d'effets atmosphériques, il y a un flou sur l'image. L'opération qui vise à rendre une image floutée plus nette s'appelle la déconvolution, et c'est ce que nous traiterons dans cette partie.

Soit l'équation initiale : $u_0 = h * u + b$. La méthode du Maximum a posteriori amène à déterminer u^* comme solution du problème :

$$min_{u \in \mathbb{R}^n} \frac{\lambda}{2} \|u_0 - (h * u)\|_2^2 + J(u)$$

5.1 Déconvolution en norme 2

Calcul du gradient et de la constante de Lipschitz

Comme on l'a vu précédemment avec le débruitage, on va calculer le gradient de la fonction Ψ où $\Psi(u) = \frac{\lambda}{2}\|u_0 - (h * u)\|_2^2 + J(u)$ afin de mettre en oeuvre un algorithme de résolution reposant sur le gradient de la fonction à minimiser.

On a

$$\nabla\Psi(u) = \lambda H^T(Hu - u_0) + \nabla J(u) \text{ avec } Hu = h * u$$

Par la suite nous utiliserons le résultat suivant : $Hu = h * u = F^{-1}[(FH).(Fu)]$ où F est la transformée de Fourier, et F^{-1} son inverse, et l'opération "." une multiplication terme à terme (FH et Fu étant des matrices diagonales, on fait le produit des composantes des diagonales terme à terme).

Ainsi, on réécrit $\nabla\Psi$:

$$\begin{aligned} \nabla\Psi(u) &= \lambda H^T(F^{-1}(FH.Fu) - u_0) + \nabla J(u) \\ &= \lambda F^{-1}((\overline{FH}).(FH).Fu - (\overline{FH}).Fu_0) + \nabla J(u) \end{aligned}$$

Remarquons que $H^* = (F^{-1}diag(FH).F)^* = F^{-1}\overline{diag(FH)}F$. FH et Fu_0 sont connus dès le départ, et calculés une fois pour toutes. En revanche, il faudra calculer Fu à chaque itération.

Calculons maintenant la constante de Lipschitz du gradient. La partie en $\nabla J(u)$ est déjà connue.

$$\lambda\|H^T(Hu - u_0) - H^T(Hv - u_0)\| = \lambda\|H^TH(u - v)\| \leq |||H^TH||||u - v\|$$

Or $H^TH = F^{-1}\overline{diag(FH)}FF^{-1}diag(FH)F = F^{-1}\overline{diag(FH)}diag(FH)F$. En passant à la norme, comme la transformée de Fourier est une isométrie, on a :

$$\|F^{-1}\overline{diag(FH)}diag(FH)F\| = \|\overline{diag(FH)}diag(FH)\|$$

Et donc $\||\overline{diag(FH)}diag(FH)|\| = sup(\overline{diag(FH)}diag(FH))$. D'où, la constante de Lipschitz de $\nabla\Psi$, notée L :

$$L = \lambda sup(\overline{diag(FH)}diag(FH)) + \frac{\||\nabla|\|^2}{\epsilon}.$$

Le pas pour l'algorithme de descente du gradient sera donc : $\tau = \frac{2}{L}$.

5.2 Déconvolution en norme 1

Cette fois, nous allons utiliser la norme L^1 plutôt que la norme L^2. Calculons le gradient de la fonction Ψ où $\Psi(u) = \lambda\|u_0 - (h * u)\|_1 + J(u)$ afin de mettre en oeuvre un algorithme de résolution basé sur le gradient de la fonction à minimiser.

Calcul du gradient et de la constante de Lipschitz

La norme 1 n'étant pas dérivable en 0, on régularise la norme $\|.\|_1$ en 0.
On utilise l'approximation $\|.\|_{1,\epsilon} \equiv \|.\|_1$ avec $\|x\|_{1,\epsilon} = \sum_{i=1}^{n}\sqrt{|x|^2+\epsilon^2}$. On a $\lim_{\epsilon\to 0}\|x\|_{1,\epsilon} = \|x\|_1$.

On a $\nabla\Psi(u) = \lambda H^T\frac{(Hu-u_0)}{\sum_i\sqrt{(Hu(i)-u_0(i))^2+\epsilon^2}} + \nabla J(u)$ avec $Hu = h * u$. Par la suite nous utiliserons le résultat suivant : $Hu = h * u = F^{-1}[(FH).(Fu)]$ où F est la transformée de Fourier, et F^{-1} son inverse, et l'opération . une multiplication terme à terme (FH et Fu étant des matrices diagonales, on fait le produit des composantes des diagonales terme à terme).

Ainsi, on réécrit $\nabla\Psi$:

$$\nabla\Psi = \lambda H^T\frac{(F^{-1}(FH.Fu) - u_0)}{\sqrt{(Hu-u_0)^2+\epsilon^2}} + \nabla J(u)$$

$$\nabla\Psi(u) = \lambda F^{-1}\frac{((\overline{FH}).(FH).Fu - (\overline{FH}).Fu_0)}{\sqrt{(Hu-u_0)^2+\epsilon^2}} + \nabla J(u)$$

De même qu'en 5.1 les termes $H^* = (F^{-1}diag(FH).F)^* = F^{-1}\overline{diag(FH)}F$. FH et Fu_0 sont connus dès le départ, et calculés une fois pour toute.

Calculons maintenant la constante de Lipschitz du gradient $\nabla\Psi(u)$. La partie en $\nabla J(u)$ est déjà connue. Par analogie avec les calculs réalisés pour la variation totale, on a :

$$L = \lambda\frac{|||H^*|||}{\epsilon} + \frac{8}{\epsilon}$$

5.2.1 Algorithme

Nous allons utiliser un algorithme de descente de gradient standard, et un algorithme de Nesterov [2]. Nous présenterons ici seulement ce dernier, la descente de gradient classique ayant été vue précédemment.

L'algorithme de Nesterov peut s'écrire de la façon suivante :
Entrée : Le nombre d'itérations N, un point de départ $x^0 \in dom(\Psi)$.
Sortie : x^N une approximation de x^*.
Début
Initialisation $A = 0$, $g = 0$, $x = x^0$
pour k allant de 0 à N **faire**
$t = 2\frac{1}{L}$
$a = \frac{t+\sqrt{t^2+4tA}}{2}$
$v = x^0 - g$
$y = \frac{Ax+av}{A+a}$
$x = y - \frac{\nabla f(y)}{L}$
$g = g + a\nabla f(x)$
$A = A + a$
Fin
Fin

avec L constante de Lipschitz de ∇f. Cet algorithme est un algorithme d'optimisation optimal, il converge plus rapidement qu'une descente de gradient classique.

5.3 Résultats

FIG. 10 – Image originale

Nous utiliserons la même image que pour le débruitage. Le critère d'arrêt sera le nombre d'itérations défini à l'avance. Les λ ont été choisis après avoir fait des recherches automatisées du λ qui minimise la quantité $\|x_\lambda - x_0\|$.

FIG. 11 – Image floutée et bruitée avec un bruit gaussien de $\sigma^2 = 0.05$, déconvolution en norme 1. Critère d'arrêt : 100 itérations, 32.3 secondes avec Matlab et 824 ms avec CUDA dont 252 ms de copie en mémoire

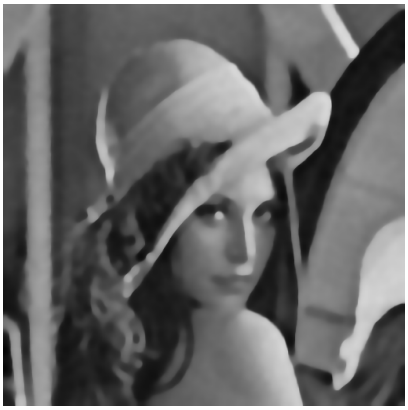

FIG. 12 – Image floutée et bruitée avec un bruit gaussien de $\sigma^2 = 0.05$, déconvolution en norme 2. Critère d'arrêt : 100 itérations, 24.1 secondes avec Matlab et 838 ms avec CUDA dont 253 ms de copie en mémoire

Déjà il est clair que la norme 1 n'est pas adapté pour faire une déconvolution avec un bruit gaussien. Nous allons maintenant tester les même algorithmes mais avec un bruit de type *Salt&Pepper*, qui va ajouter des points noirs et des points blancs aléatoirement sur l'image.

FIG. 13 – Image floutée et bruitée avec un bruit Salt&pepper, la moitié des pixels étant mis à 1 ou à 0. Déconvolution en norme 1. Critère d'arrêt : 100 itérations, 35.4 secondes avec Matlab et 870 ms avec CUDA dont 253 ms de copie en mémoire

Nous allons utiliser une image plus grosse, venant d'un appareil photo 10M pixels. Le critère d'arrêt sera le nombre d'itérations défini à l'avance. Les λ ont été choisis

après avoir fait des recherches automatisées du λ qui minimise la quantité $\|x_\lambda - x_0\|$.

FIG. 14 – Image originale

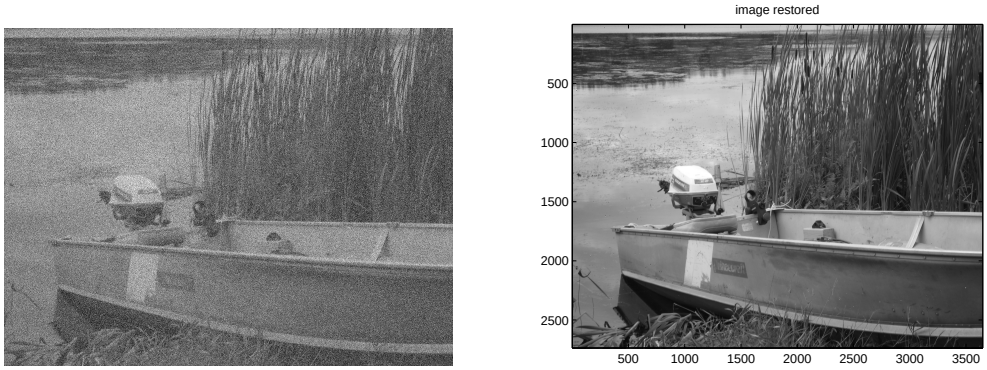

FIG. 15 – Image floutée et bruitée avec un bruit Salt&Pepper, la moitié des pixels étant mis à 1 ou à 0. Déconvolution en norme 1. Critère d'arrêt : 100 itérations, 24 min avec Matlab et 4 min avec CUDA

FIG. 16 – Image floutée et bruitée avec un bruit Salt&Pepper, la moitié des pixels étant mis à 1 ou à 0. Déconvolution en norme 2. Critère d'arrêt : 100 itérations, 21 min avec Matlab et 4 min avec CUDA

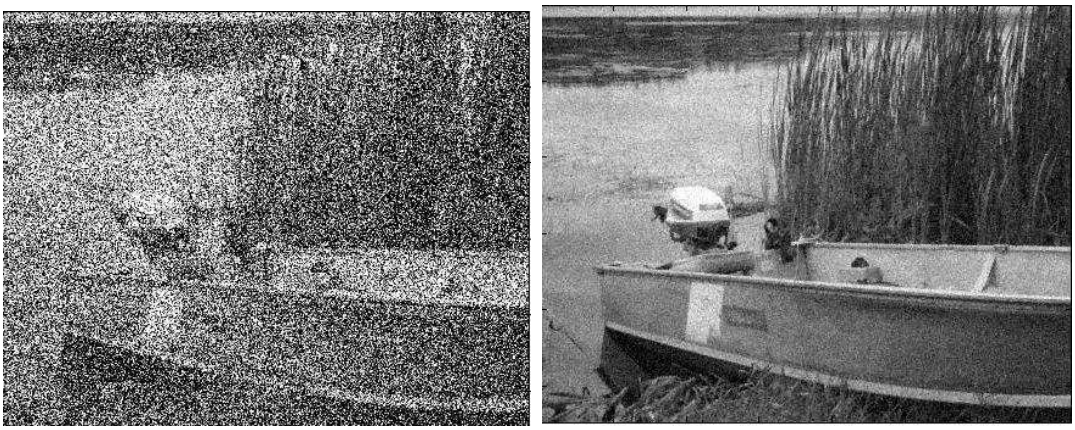

FIG. 17 – Image floutée et bruitée avec un bruit gaussien de $\sigma^2 = 0.5$, déconvolution en norme 1. Critère d'arrêt : 100 itérations, 25 min avec Matlab et 5 min avec CUDA

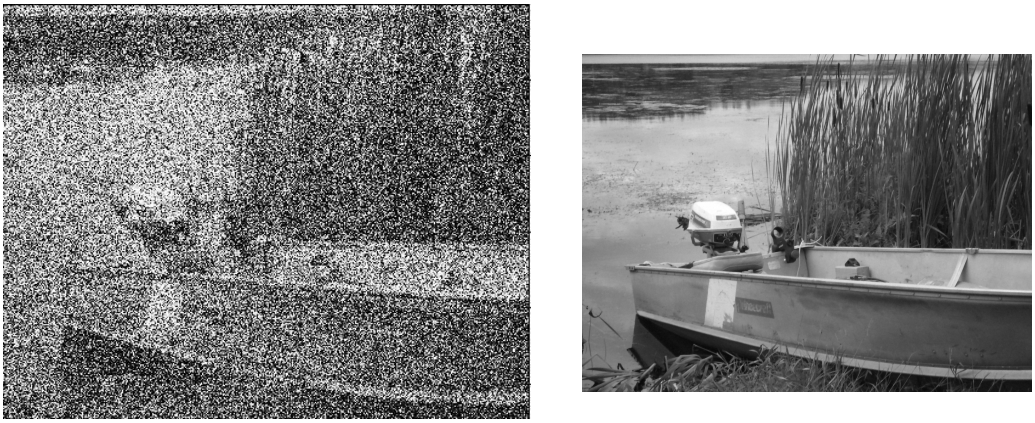

FIG. 18 – Image floutée et bruitée avec un bruit gaussien de $\sigma^2 = 0.5$, déconvolution en norme 2. Critère d'arrêt : 100 itérations, 19 min avec Matlab et 4 min avec CUDA

Déjà il est clair que la norme 1 n'est pas adapté pour faire une déconvolution avec un bruit gaussien. Nous allons maintenant tester les même algorithmes mais avec un bruit de type *Salt&Pepper*, qui va ajouter des points noirs et des points blancs aléatoirement sur l'image.

On voit que la norme 1 donne un meilleur résultat pour ce type de bruit, même si le temps de calcul est légèrement plus long. Il est à noter que l'image originale était fortement dégradée : le résultat de la restauration est impressionnant, mais on peut encore obtenir mieux en affinant le choix des paramètres. En utilisant CUDA, on divise le temps de calcul par 50 environ, sachant que sur ce temps là, à peu près un quart est du aux accès à la mémoire. De plus nous pensons que le code peut encore être amélioré pour gagner en temps avec une gestion de la mémoire plus fine.

Pour une image plus grosse on constate que le gain en temps diminue (5 à 6 plus rapide seulement), cela est du au fait que l'on ne peut plus traiter une image aussi grande en une seule fois en utilisant seulement la mémoire du GPU. On stocke donc dans la RAM du système grâce à des allocations de mémoires spéciales, mais on ralentit en même temps énormement le calcul. Une autre solution est de diviser l'image en blocs tels qu'il puissent être traités en étant chargé entièrement sur le GPU, les différents blocs étant traités en séries. Cette méthode laisse présager des gains en temps de calcul de l'ordre de 50.

6 Conclusion

Au cours de ce projet nous avons étudier des methodes de restaurations d'images brouillées et bruitées. Ce type de problème est souvent rencontré en imagerie en astronomie, où les miroirs imparfaits et les effets atmosphériques brouillent les images. Les opérations de débruitage et de déconvolution sont des opérations lourdes en calculs, et les problèmes d'optimisation croissent en même temps que la taille de l'image, ce qui pose des problèmes de temps de calcul (et parfois de stockage aussi : une image d'un apparail photo 10M pixel est une variable pesant plus de 80Mo quand on utilise la double précision !). La technologie CUDA de NVIDIA nous a permis de concevoir des algorithmes de résolution parallèle pour ces problèmes, et les gains en temps de calcul sont au rendez-vous. Avec un GPU grand public, nous avons observé des gains de performance parfois supérieur à 130 par rapport à Matlab, tout en sachant que dans ces cas là près d'un tiers du temps d'execution correspond à la copie des variables dans la mémoire du GPU, le temps effectif de calcul étant extrêment court.

CUDA est une technologie encore jeune, mais qui du fait de son énorme potentiel progresse très vite. On parle déjà d'OpenCL, une API C fortement inspiré de CUDA et multiplateforme (qui supporte aussi bien les GPU AMD/ATI que ceux de NVIDIA), qui permettrait d'ouvrir encore plus de perspectives au calcul sur GPU.

Pour finir, ce projet a été source d'enrichissement personnel, il nous a permis de nous ouvrir au traitement d'image, à la programmation parallèle et nous a ouvert de nombreuses opportunités. Nous avons décidé de poursuivre dans cette voie avec nos stages respectifs.

Nous tenons à remercier Aude RONDEPIERRE et Pierre WEISS pour leur aide, leurs explications, et pour nous avoir guidé tout au long de ce projet.

7 Annexes

7.1 Quelques codes

Voici deux codes pour la déconvolution, le premier sous Matlab, le deuxième sous CUDA (le main.c et le Deconvolution.cu, le fichier où se trouve le calcul à proprement parler). Il s'agit de la déconvolution avec la norme 2. Nous avons choisis de mettre ces codes car ce sont les plus complets et le mécanisme du débruitage y est inclus.

Les codes CUDA ont été compilés sous LINUX (sur une distribution Ubuntu 9.04 64 bits), avec la version 2.3 du SDK de Nvidia : `http://www.nvidia.fr/object/cuda_learn_fr.html`.

Les résultats en temps de calculs ont été obtenus sur cette même machine équipée d'un processeur Intel Core 2 Duo E8400 (2*3Ghz), de 4Go de ram et d'un GPU Nvidia GTX 260 avec de 898MB de ram.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%Projet INSA GMM 5A
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%Déconvolution d'une image

clear all;
close all;
hold off;

tic
%Chargement de l'image
%load lenar.mat;
%load map;
%colormap(map);
IM=imread('lena1.jpg');
IM=im2double(IM(:,:,1));
%IM=IM(1:4:end,1:4:end);
figure(1);
%image(IM);
imshow(IM);

[I J]=size(IM);

%Creation du filtre
%Taille du filtre m
m=5;
h=zeros(size(IM));
h(1:m,1:m)=1/(m^2);


%fft h une fois pour toutes
H=fft2(h);

%Convolution
IM_C= real(ifft2(fft2(IM).*H));
```

```
37
38 %Perturbation
39 %IM_C=IM_C+0.05*randn(I,J);
40 IM_C = imnoise(IM_C,'salt & pepper', 0.1);
41
42 figure(2);
43 imshow(IM_C);
44
45 %Deconvolution
46
47 %Paramètre de la descente de gradient
48 Nbre_Iter=200;
49 lambda=0.1;
50 Epsilon=0.1;
51
52 %Calcul du pas
53 L=max(max((conj(H).*H)));
54 Lipschitz=(lambda*L + 8/Epsilon);
55 Pas=2/(lambda*L + 8/Epsilon);
56 l=0;%2*max(max(conj(H).*H));
57
58 FU0=fft2(IM_C);
59
60 x=IM_C;
61 x0=IM_C;
62 A=0;
63 g=0;
64
65 %Descente de gradient
66 wait=waitbar(0);
67 for n=1:Nbre_Iter
68
69       %iteration avec la descente de gradient améliorée
70       t=2*(1+l*A)/Lipschitz;
71       a=(t+sqrt(t^2+4*t*A))/2;
72       v=IM_C-g;
73       y=(A*x+a*v)/(A+a);
74       %Calcul du Grad(y)
75       %Calcul de DJ(y)
76       [G1 G2]=grad(y);
77       Nij=sqrt(G1.^2+G2.^2+Epsilon^2);
78       G1=G1./Nij;
79       G2=G2./Nij;
80
81      D=div(G1,G2);
82
83      %Calcul de la partie en h et assemblage de Grad(y)
84      Grad_Psiy=lambda*real(ifft2(conj(H).*H.*fft2(y)-conj(H).*FU0))- D;
85
86       x=y-Grad_Psiy/Lipschitz;
87       [G1 G2]=grad(x);
88       Nij=sqrt(G1.^2+G2.^2+Epsilon^2);
89       G1=G1./Nij;
90       G2=G2./Nij;
91
```

```
92      D=div(G1,G2);
93
94      %Calcul de la partie en h et assemblage de Grad(y)
95      Grad_Psix=lambda*real(ifft2(conj(H).*H.*fft2(x)-conj(H).*FU0))- D;
96
97       g=g+a*Grad_Psix;
98       A=A+a;
99       ng=sqrt(G1.^2+G2.^2+Epsilon^2);ngh=sqrt((real(ifft2(H.*fft2(x))-x0)).^2 +
             Epsilon^2);
100      Costn2(n)=sum(ng(:)+lambda*ngh(:));
101      if(mod(n,10)==0)
102          figure(4);
103          colormap(gray);
104          imagesc(x);
105          Costn2(n)
106          title(sprintf('iteration %i/%i',n,Nbre_Iter))
107     end
108
109      waitbar(n/Nbre_Iter,wait);
110 end
111
112
113 figure(3);
114 imshow(x);
115
116
117
118 toc
```

Listing 1 – main.cpp

```
/*****************************************************************************
----------Programme de descente du gradient
    ------------------------------

-----------------------------------------------------------------------------

*****************************************************************************/

//
    -----------------------------------------------------------------------------

// Inclusions

        // Librairies C
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

        // Librairies CUDA
#include <cuda_runtime.h>
#include <cutil.h>
#include <cufft.h>

        //Fichiers personnels

//
    -----------------------------------------------------------------------------

// Declarations externes

        // Fonction affichant les caracteristiques des cartes installees sur la machine
        extern "C" void Devices_properties();

        // Fonction permettant la resolution par algorithme sequentiel sur CPU, et armoï¿½age lors
            de la resolution en parallele
        //extern "C" void Descente_gradient_cpu( double*, int, int );

        // Fonction permettant la resolution par algorithme parallel sur GPU
        extern "C" void Deconvolution_gpu( double, double*, double*, int, int );

        // Function permettant de verifier le bon deroulement des appels aux fonctions CUDA
        extern "C" void Check_CUDA_error(const char* msg);

//
    -----------------------------------------------------------------------------

// Declarations de fonctions utilisees par la suite et definies en fin de fichier

        // Fonction permettant de creer et initialiser les timers sur gpu
        void Creer_timer_gpu();

//
    -----------------------------------------------------------------------------
```

```
// Declarations des variables externes

47          // Variables permettant le calcul du temps ecoule a l'interieur d'une fonction
            double time_total_gpu, time_kernel, time_memcpy, timer_init;

            // Variables permettant de stocker les informations de la configuration de la carte utilisee
            int SHARED_MEMORY_SIZE_PER_BLOCK;
52          int MAX_NUM_THREADS_PER_BLOCK;
            int MAX_NUM_BLOCKS;
            int WARP_SIZE;
            int GRID_DIM;
            int BLOCK_DIM;
57
            // Variables permettant de connaitre le temps d'execution de differentes parties du
                programme parallel
            unsigned int timer_total_gpu, timer_kernel, timer_memcpy;


62  /*
    ----------------------------------------------------------------------------

    Programme principal
    ----------------------------------------------------------------------------
        */

    int main(int argc, char **argv)
67  {
            // Affichage des caracteristiques des cartes installees sur la machine:
            // la carte 0 est la carte utilisee par la suite
            Devices_properties();

72          printf("\n\n\n\n\n\n\n\n\t\t\t Deconvolution:\n\n");

        // Initialisation du device (GPU)
            CUT_DEVICE_INIT(argc, argv);
            Check_CUDA_error("Initialisation du device");
77
            //Creation des timer_cpu
            //clock_t timer_init, timer_fin_sequentiel, timer_total_sequentiel, timer_fin_programme,
                timer_total_programme;
            double time_total_sequentiel = 0;
            double time_total_programme = 0;
82          // Creation des timer_gpu
            Creer_timer_gpu();

        // Initialisation du temps calcul sequentiel
            timer_init =clock();
87
            printf("\nChargement du fichier source image\n");
            // Chargement de l'image IM sous la forme d'un pointeur vers des double
            int hIM;
            int lIM;
92          hIM=lIM=0;
            FILE * MonFichier;
```

```
MonFichier = fopen(argv[1], "r");
if (MonFichier==NULL){
        printf ("\nLe fichier n' existe pas\n");
}
else{
        printf ("\nLe fichier source a bien ete ouvert\n");
}

fscanf(MonFichier, "%d %d\n", &hIM, &lIM);
double IM [hIM*lIM];
for(int i=0;i<hIM*lIM;i++){
        fscanf(MonFichier, "%lf", &IM[i]);
}

fclose (MonFichier);
printf ("Fichier image source charge avec succes\n");

printf ("Chargement du fichier h\n");
// Chargement de H sous la forme d'un pointeur vers des double
double L;
L=0.0;
FILE * MonFichierH;
MonFichierH = fopen(argv[2],"r");
if (MonFichierH==NULL){
        printf ("Le fichier n' existe pas\n");
}
else{
        printf ("Le fichier source a bien ete ouvert\n");
}

fscanf(MonFichierH, "%lf\n", &L);
double H [hIM*lIM];
for(int i=0;i<hIM*lIM;i++){
        fscanf(MonFichier, "%lf", &H[i]);
}

fclose (MonFichier);
printf ("Fichier H source charge avec succes\n");


// Deformation de l'image IM ==>Deja fait dans l'impage recuperee dans Matlab
// Definition de la hauteur hIM et de la largeur lIM de l'image==>Deja fait

// Depart du timer pour calculer le temps d'execution du prgm sur GPU
CUT_SAFE_CALL(cutStartTimer(timer_total_gpu));

// Appel au programme qui gere la resolution sur GPU
printf ("Lancemenent de la deconvolution sur GPU\n");
Deconvolution_gpu(L, H, IM, hIM, lIM );
printf ("Descente de gradient terminee\n");
// Synchronisation pour s'assurer que tous les threads ont termine
CUDA_SAFE_CALL(cudaThreadSynchronize());

// Stop timer
CUT_SAFE_CALL(cutStopTimer(timer_total_gpu));
```

```
        // Gestion des erreurs
        Check_CUDA_error("Fin algorithme GPU");

        //for(int i=0;i<10;i++)
        //      printf("%f\n",lM(i));

        FILE * MonFichierMod;
        MonFichierMod = fopen(argv[3],"w");
        printf("Fichier de resultat ouvert avec succes\n");
        fprintf(MonFichierMod, "%d\n",hIM);
        fprintf(MonFichierMod,"%d\n", lIM);

        for(int i=0;i<(hIM*lIM);i++) {
                fprintf(MonFichierMod, "%lf\n", IM[i]);
        }
        fclose(MonFichierMod);

        //for(int i=0;i<(18*18);i++)
        //      printf("%f\n", sub_d(i));

        // Affichage apres conversion du temps necessaire
        time_total_gpu                  = cutGetTimerValue( timer_total_gpu );
        time_kernel                     = cutGetTimerValue( timer_kernel );
        time_memcpy                     = cutGetTimerValue(timer_memcpy);

        printf("time total gpu %lf\n time kernel %lf\n time memcpy %lf\n",time_total_gpu, time_kernel,
            time_memcpy);

    // Prompt final
        printf("\n");
    if (argc == 1)
        CUT_EXIT(argc, argv);
}

//
    ------------------------------------------------------------------------

// Fonction permettant de creer et initialiser les timers sur gpu

void Creer_timer_gpu()
{
        CUT_SAFE_CALL(cutCreateTimer(&timer_total_gpu));
        CUT_SAFE_CALL(cutCreateTimer(&timer_kernel));
        CUT_SAFE_CALL(cutCreateTimer(&timer_memcpy));

        CUT_SAFE_CALL(cutResetTimer(timer_total_gpu));
        CUT_SAFE_CALL(cutResetTimer(timer_kernel));
        CUT_SAFE_CALL(cutResetTimer(timer_memcpy));
}
```

Listing 2 – Deconvolution_amelioree.cu

```
/*****************************************************************************
------------------------------Deconvolution
    //------------------------------

------------------------------------------------------------------------------

*****************************************************************************/


//
    ------------------------------------------------------------------------------

// Inclusions

        // Librairies C
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>

        // Librairies CUDA
#include <cutil.h>
#include <cuda.h>
#include <cufft.h>

        // Fichiers personnels


//
    ------------------------------------------------------------------------------

// Variables automatiques

#define BLOCK_SIZE 16


//
    ------------------------------------------------------------------------------

// Declarations de fonctions utilisees par la suite et definies en fin de fichier

        // Les Kernels utilises
__global__ void kernel_Grad(cufftDoubleComplex*, int, int, double, double, double, cufftDoubleComplex *,
    double * );
        // Fonctions externes:
extern "C" void Check_CUDA_error( const char* );
__global__ void Conjuguee(cufftDoubleComplex*, cufftDoubleComplex*, int, int);
__global__ void CalculTransF(cufftDoubleComplex*, cufftDoubleComplex*, cufftDoubleComplex*,
    cufftDoubleComplex*, int, int);
__global__ void realtocomplex(double *, cufftDoubleComplex *, int, int);
__global__ void complextoreal(cufftDoubleComplex *, double *, int, int);
__global__ void putinUk_d (cufftDoubleComplex *, double *, int);
__global__ void CalculY (cufftDoubleComplex *, double *, double, double, double *, cufftDoubleComplex *,
    int);
```

```
__global__ void Calculx (cufftDoubleComplex *, cufftDoubleComplex *, double *, double, int);
__global__ void Calculg (double *, double *, double, int);
__global__ void Initg (double *, int);

48
        // Fonctions internes:


//
    ------------------------------------------------------------------------

53  // Variables externe:

extern int GRID_DIM;
extern int BLOCK_DIM;
extern int WARP_SIZE;
58

//
    ------------------------------------------------------------------------

// Fontion qui se deroule sur le host (CPU) et gere le travail sur le device (GPU)
//
    ------------------------------------------------------------------------

63
extern "C" void Deconvolution_gpu (double L, double* H, double* IM, int hIM, int lIM )
{
        // Declarations des variables utilisees pour la gestion du temps
        extern unsigned int timer_kernel;
68      extern unsigned int timer_memcpy;
        cudaDeviceProp deviceProp;

        cudaGetDeviceProperties(&deviceProp,0);
        if (!deviceProp.canMapHostMemory){
73              exit (EXIT_FAILURE);
        }

        // Declarations des variables utilisees pour la resolution
        int Nbr_iter = 100;
78      double Epsilon = 0.05;
        double Lambda = 20;
        double Pas = 2/(Lambda*L + 8/Epsilon);
        double Lipschitz=Lambda*L + 8/Epsilon;
        double l = 0; //Constance de force convexite
83
        // Variables
        double* Uk_d;
        double * H_d;
        cufftDoubleComplex* CUk_d;
88      cufftDoubleComplex* FUo_d;
        cufftDoubleComplex* CH_d;
        cufftDoubleComplex* CGrad;
        cufftHandle plan;
        int size = hIM * lIM;
```

```
93
        //Variables specifiques a la descente amelioree
        double A=0;
        double a;
        double t;
98      double * Uo;
        double * g;
        double * Grady;
        double * Gradx;
        cufftDoubleComplex* Cy;
103     cufftDoubleComplex* Fy;

        //mapped variables
        /*cufftDoubleComplex* CGrad_m;
        double * g_m;
108     double * Grady_m;
        double * Gradx_m;
        cufftDoubleComplex* Cy_m;
        cufftDoubleComplex* Fy_m;
        cufftDoubleComplex* FUo_d_m;
113     cufftDoubleComplex* CH_d_m;
        CUDA_SAFE_CALL(cudaSetDeviceFlags(cudaDeviceMapHost));
        Check_CUDA_error("cudaSetDeviceFlags");*/

        // Allocations
118      printf (" Allocations des variables en memoire\n");
        CUDA_SAFE_CALL(cudaMalloc((void**)&Uk_d, sizeof(double) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&CUk_d, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&H_d, sizeof(double) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&Uo, sizeof(double) *size));
123
        CUDA_SAFE_CALL(cudaMalloc((void**)&CH_d, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&CGrad, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&g, sizeof(double) *size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&Grady, sizeof(double) *size));
128     CUDA_SAFE_CALL(cudaMalloc((void**)&Gradx, sizeof(double) *size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&Cy, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&Fy, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&FUo_d, sizeof(cufftDoubleComplex) * size));
        CUDA_SAFE_CALL(cudaMalloc((void**)&Resultat, sizeof(double) * size));
133
        //Allocation on Host mapped memory
        /*CUDA_SAFE_CALL(cudaHostAlloc((void**)&CH_d_m, sizeof(cufftDoubleComplex) * size,
              cudaHostAllocMapped));
        CUDA_SAFE_CALL(cudaHostAlloc((void**)&CGrad_m, sizeof(cufftDoubleComplex) * size,
              cudaHostAllocMapped));
        CUDA_SAFE_CALL(cudaHostAlloc((void**)&g_m, sizeof(double) *size,
              cudaHostAllocMapped));
138     CUDA_SAFE_CALL(cudaHostAlloc((void**)&Grady_m, sizeof(double) *size,
              cudaHostAllocMapped));
        CUDA_SAFE_CALL(cudaHostAlloc((void**)&Gradx_m, sizeof(double) *size,
              cudaHostAllocMapped));
        CUDA_SAFE_CALL(cudaHostAlloc((void**)&Cy_m, sizeof(cufftDoubleComplex) * size,
              cudaHostAllocMapped));
```

```
CUDA_SAFE_CALL(cudaHostAlloc((void**)&Fy_m, sizeof(cufftDoubleComplex) * size,
    cudaHostAllocMapped));
CUDA_SAFE_CALL(cudaHostAlloc((void**)&FUo_d_m, sizeof(cufftDoubleComplex) * size,
    cudaHostAllocMapped));
Check_CUDA_error("cudaHostAllocDefault");

//Pointeur du device sur la mapped memory
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&CH_d,(void*)CH_d_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&CGrad,(void*)CGrad_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&g,(void*)g_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&Grady,(void*)Grady_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&Gradx,(void*)Gradx_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&Cy,(void*)Cy_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&Fy,(void*)Fy_m,0));
CUDA_SAFE_CALL(cudaHostGetDevicePointer((void**)&FUo_d,(void*)FUo_d_m,0));
Check_CUDA_error("cudaHostGetDevicePointer");*/

// Transferts initiaux de donnees
CUT_SAFE_CALL(cutStartTimer(timer_memcpy));
CUDA_SAFE_CALL(cudaMemcpy(Uk_d, IM, sizeof(double)*size, cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(Uo, IM, sizeof(double)*size, cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(H_d, H, sizeof(double)*size, cudaMemcpyHostToDevice));
Check_CUDA_error("cudaMemcpy");

// Dimensions des blocks et de la grille
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid (lIM/dimBlock.x, hIM/dimBlock.y);

//convertion
realtocomplex<<<dimGrid, dimBlock>>>(Uk_d,CUk_d,lIM,hIM);
realtocomplex<<<dimGrid, dimBlock>>>(H_d,CH_d,lIM,hIM);
Check_CUDA_error("conversions en complexes");

// CrÃ©ation du plan
cufftPlan1d(&plan, hIM*lIM, CUFFT_Z2Z, 1);
cufftExecZ2Z(plan, CH_d, CH_d, CUFFT_FORWARD);
Check_CUDA_error("FFT de H");

// Transferts initiaux de donnees
CUT_SAFE_CALL(cutStopTimer(timer_memcpy));
Check_CUDA_error("Copies initiales");

// Initialisation
//CUDA_SAFE_CALL(cudaFree(H_d));
//CUDA_SAFE_CALL(cudaFree(CH_d));
Initg <<<dimGrid, dimBlock>>>(g, lIM);
cufftExecZ2Z(plan, CUk_d, FUo_d, CUFFT_FORWARD);
Check_CUDA_error("FFT Uk_d");

// Calculs
printf("Debut des calculs \n");
CUT_SAFE_CALL(cutStartTimer(timer_kernel));
printf("Debut des itÃ©ration\n");
for (int iter = 0; iter < Nbr_iter; iter ++)
{
```

```
                t=2*(1+l*A)/(Lambda*L + 8/Epsilon);
                a=(t+sqrt(pow(t,2)+4*t*A))/2;
                CalculY<<<dimGrid, dimBlock>>>(CUk_d, Uo, A, a, g, Cy, lIM);
                cufftExecZ2Z(plan, Cy, Fy, CUFFT_FORWARD);
198             CalculTransF <<<dimGrid, dimBlock>>> (CH_d, FUo_d, Fy, CGrad, lIM, hIM);
                cufftExecZ2Z(plan, CGrad, CGrad, CUFFT_INVERSE);
                //Penser a diviser le resultat par le nbre d'elements
                kernel_Grad <<< dimGrid, dimBlock >>> (Cy, hIM, lIM, Epsilon, Pas, Lambda, CGrad, Grady);
                Calculx<<<dimGrid,dimBlock>>>(CUk_d, Cy, Grady, Lipschitz, lIM);
203             cufftExecZ2Z(plan, CUk_d, CUk_d, CUFFT_FORWARD);
                CalculTransF <<<dimGrid, dimBlock>>> (CH_d, FUo_d, CUk_d, CGrad, lIM, hIM);
                cufftExecZ2Z(plan, CGrad, CGrad, CUFFT_INVERSE);
                cufftExecZ2Z(plan, CUk_d, CUk_d, CUFFT_INVERSE);
                //Penser a diviser le resultat par le nbre d'elements
208             kernel_Grad <<< dimGrid, dimBlock >>> (CUk_d, hIM, lIM, Epsilon, Pas, Lambda, CGrad,
                    Gradx);
                Calculg<<<dimGrid, dimBlock>>>(g, Gradx, a, lIM);
                A=a+A;
            }
            Check_CUDA_error("Fin des iterations");
213          printf ("Fin des iteration \n");
            CUT_SAFE_CALL(cutStopTimer(timer_kernel));
             printf ("Arret du timer\n");
            CUT_SAFE_CALL(cutStartTimer(timer_memcpy));
             printf ("Copie de Uk_d sur le host\n");
218         CUDA_SAFE_CALL(cudaMemcpy(IM, Uk_d, sizeof(double)*size, cudaMemcpyDeviceToHost));
            CUT_SAFE_CALL(cutStopTimer(timer_memcpy));
            //Check_CUDA_error("Copie finale");

    }
223
    //
        ------------------------------------------------------------------------

    // Kernel permettant d'iterer sur Uk_d

    __global__ void kernel_Grad(cufftDoubleComplex* CUk_d, int hIM, int lIM, double Epsilon, double Pas,
        double Lambda, cufftDoubleComplex* CGrad, double* GradPsixy)
228 {
            // Configuration
            int bx = blockIdx.x;
            int by = blockIdx.y;
            int tx = threadIdx.x;
233         int ty = threadIdx.y;

            // Index des deux sous matrices utilisees par le block
            int begin = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;

238         // Element de la matrice Gradient de Psi gere par le thread
            double gradPsi;
            gradPsi =0.0;

            // Declaration des deux sous matrices utilisees par le block
243         __shared__ double subUk_d[BLOCK_SIZE+2][BLOCK_SIZE+2];
```

```
// Chargement des donnees depuis les matrices en global_mem vers les subMatrices
    en sharedmem
// Chaque thread du block charge un element pour les deux subMatrices
subUk_d[ty+1][tx+1] = CUk_d[begin + lIM * ty + tx ].x;

if (((tx+ty*BLOCK_SIZE)/warpSize) == 0 ) //on requisitionne les 32 premiers threads
{
    if ( by!=0 && by!=(gridDim.y - 1) && bx!=0 && bx!=(gridDim.x - 1) ) //sous-matrice
        non au bord
    {
        subUk_d[ty * (BLOCK_SIZE + 1)][tx+1] = CUk_d[begin + lIM*( (BLOCK_SIZE + 1)*ty -
            1) + tx].x; //on charge les lignes en haut et en bas. tx varie de 0 a 15, et ty
            = 0 ou 1
        subUk_d[tx+1][ty * (BLOCK_SIZE + 1)] = CUk_d[begin + (BLOCK_SIZE + 1)*ty - 1 + tx
            * lIM].x; //on charge les colonnes de g. et d. !!!Mauvais
    }
    else if ( by == 0 )
    {
        if ( bx!=0 && bx!=(gridDim.x - 1) ) //sous-matrice superieures non au bord
        {
            subUk_d[tx+1][ty * (BLOCK_SIZE + 1)] = CUk_d[begin + (BLOCK_SIZE + 1)*ty -
                1 + tx * lIM].x; //on charge les colonnes de g. et d. !!!Mauvais
            if ( ty == 0 )
                subUk_d[BLOCK_SIZE + 1][tx+1] = CUk_d[begin + lIM * BLOCK_SIZE + tx].x;
                    //on charge la ligne en bas. tx varie de 0 a 15, et ty = 0 ou 1
                                    }
        }
        else if ( bx == 0 ) //sous_matrice coin sup g.
        {
            if (ty == 0 )
                subUk_d[tx+1][BLOCK_SIZE + 1] = CUk_d[begin + BLOCK_SIZE + tx * lIM].x; //
                    on charge la colonne de d. !!!Mauvais
            else
                subUk_d[BLOCK_SIZE + 1][tx+1] = CUk_d[begin + lIM * BLOCK_SIZE + tx].x;
                    //on charge la ligne en bas. tx varie de 0 a 15, et ty = 0 ou 1
        }
        else if ( bx == (gridDim.x - 1) ) //sous-matrice coin sup d.
        {
            if (ty == 0 )
                subUk_d[tx+1][0] = CUk_d[begin - 1 + tx * lIM].x; //on charge la colonne
                    de g. !!!Mauvais
            else
                subUk_d[BLOCK_SIZE + 1][tx+1] = CUk_d[begin + lIM * BLOCK_SIZE + tx].x; //
                    on charge la ligne en bas. tx varie de 0 a 15, et ty = 0 ou 1
        }
    }
    else if ( by == (gridDim.y - 1) )
    {
        if ( bx!=0 && bx!=(gridDim.x - 1) ) //sous-matrice infi¿½rieures non au bord
        {
            subUk_d[tx+1][ty * (BLOCK_SIZE + 1)] = CUk_d[begin + (BLOCK_SIZE + 1)*ty - 1
                + tx * lIM].x; //on charge les colonnes de g. et d. !!!Mauvais
            if ( ty == 0 )
                subUk_d[0][tx+1] = CUk_d[begin - lIM + tx].x; //on charge la ligne en haut.
                    tx varie de 0 a 15, et ty = 0 ou 1
```

```
            }
            else if ( bx == 0 ) //sous_matrice coin inf g.
288         {
                if (ty == 0 )
                    subUk_d[tx+1][BLOCK_SIZE + 1] = CUk_d[begin + BLOCK_SIZE + tx * lIM].x; //
                        on charge la colonne de d. !!!Mauvais
                else
                    subUk_d[0][tx+1] = CUk_d[begin - lIM + tx ].x; //on charge la ligne en
                        haut. tx varie de 0 a 15, et ty = 0 ou 1
293         }
            else if ( bx == (gridDim.x - 1) ) //sous-matrice coin inf d.
            {
                if ( ty == 0 )
                    subUk_d[tx+1][0] = CUk_d[begin - 1 + tx * lIM].x; //on charge la colonne
                        de g. !!!Mauvais
298             else
                    subUk_d[0][tx+1] = CUk_d[begin - lIM + tx].x; //on charge la ligne en haut.
                         tx varie de 0 a 15, et ty = 0 ou 1
            }
        }
        else if ( bx == 0 && by != 0 && by != (gridDim.y - 1) ) //sous-matrices de g. non au
            bord
303     {
            subUk_d[ty * (BLOCK_SIZE + 1)][tx+1] = CUk_d[begin + lIM*( (BLOCK_SIZE + 1)*ty -
                1) + tx].x; //on charge les lignes en haut et en bas. tx varie de 0 a 15, et ty =
                 0 ou 1
            if ( ty == 0 )
                subUk_d[tx+1][BLOCK_SIZE + 1] = CUk_d[begin + BLOCK_SIZE + tx * lIM].x; //on
                    charge la colonne de d. !!!Mauvais
        }
308     else if ( bx == (gridDim.x - 1) && by != 0 && by != (gridDim.y - 1) ) //sous-matrices
            de d. non au bord
        {
            subUk_d[ty * (BLOCK_SIZE + 1)][tx+1] = CUk_d[begin + lIM*( (BLOCK_SIZE + 1)*ty -
                1) + tx].x; //on charge les lignes en haut et en bas. tx varie de 0 a 15, et ty =
                 0 ou 1
            if ( ty == 0 )
                subUk_d[tx+1][0] = CUk_d[begin - 1 + tx * lIM].x; //on charge la colonne de g.
                    !!!Mauvais
313     }
    }

    // Synchronisation pour s'assurer que les subMatrices sont chargees
    __syncthreads () ;
318

    if ( by == 0 && ty == 0) //sous matrices sup, lignes sup
    {
        if ( bx == 0 && tx == 0 ) // i=1 et j=1
            gradPsi = gradPsi - ( (subUk_d[2][1]-subUk_d[1][1]) + (subUk_d[1][2]-subUk_d[1][1]) )
                *rsqrt( pow((subUk_d[2][1]-subUk_d[1][1]),2) + pow((subUk_d[1][2]-subUk_d[1][1])
                ,2) + pow(Epsilon,2) );
323     else if ( bx == (gridDim.x - 1) && tx == (BLOCK_SIZE - 1) ) //i=1 et j=J
            gradPsi = gradPsi - (subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1])*rsqrt(pow((subUk_d[ty
                +2][tx+1]-subUk_d[ty+1][tx+1]),2)+pow(Epsilon,2));
        else
```

```
                gradPsi = gradPsi - ( (subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1]) + (subUk_d[ty+1][tx
                    +2]-subUk_d[ty+1][tx+1]) )*rsqrt(pow((subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1])
                    ,2)+pow((subUk_d[ty+1][tx+2]-subUk_d[ty+1][tx+1]),2)+pow(Epsilon,2)) + (subUk_d[
                    ty+1][tx+1]-subUk_d[ty+1][tx])*rsqrt(pow((subUk_d[ty+2][tx]-subUk_d[ty+1][tx]),2)+
                    pow((subUk_d[ty+1][tx+1]-subUk_d[ty+1][tx]),2)+pow(Epsilon,2));

328         }
            else if ( by == (gridDim.y - 1) && ty == (BLOCK_SIZE - 1) ) //sous-matrices inf, lignes inf
            {
                if ( bx == 0 && tx == 0 ) // i=1 j=1
                gradPsi = gradPsi - (subUk_d[ty+1][tx+2]-subUk_d[ty+1][tx+1])*rsqrt(pow((subUk_d[ty
                    +1][tx+2]-subUk_d[ty+1][tx+1]),2)+pow(Epsilon,2));
333             else if ( bx != (gridDim.x - 1) || tx != (BLOCK_SIZE - 1) ) // i=1 j!=J
                gradPsi = gradPsi - (subUk_d[ty+1][tx+2]-subUk_d[ty+1][tx+1])*rsqrt(pow((subUk_d[ty
                    +1][tx+2]-subUk_d[ty+1][tx+1]),2)+pow(Epsilon,2)) + (subUk_d[ty+1][tx+1]-subUk_d
                    [ty+1][tx])*rsqrt(pow((subUk_d[ty+1][tx+1]-subUk_d[ty+1][tx]),2)+pow(Epsilon,2));
            }
            else if ( bx == 0 && tx == 0 ) //j=1 et le sinon => i!=1 et i!=1
            {
338             gradPsi = gradPsi - (subUk_d[ty+2][tx+1]-2*subUk_d[ty+1][tx+1]+subUk_d[ty+1][tx+2])*rsqrt
                    (pow((subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1]),2)+pow((subUk_d[ty+1][tx+2]-subUk_d
                    [ty+1][tx+1]),2)+pow(Epsilon,2)) + (subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1])*rsqrt(pow((
                    subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1]),2)+pow((subUk_d[ty][tx+2]-subUk_d[ty][tx+1])
                    ,2)+pow(Epsilon,2));
            }
            else if ( bx == (gridDim.x - 1) && tx == (BLOCK_SIZE - 1) ) //j=J et i!=1 et i!=I
            {
                gradPsi = gradPsi - (subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1])*rsqrt(pow((subUk_d[ty+2][tx
                    +1]-subUk_d[ty+1][tx+1]),2)+pow(Epsilon,2)) + (subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1])
                    *rsqrt(pow((subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1]),2)+pow(Epsilon,2));
343         }
            else
            {
                gradPsi = gradPsi - (subUk_d[ty+2][tx+1]+subUk_d[ty+1][tx+2]-2*subUk_d[ty+1][tx+1])*rsqrt
                    (pow((subUk_d[ty+2][tx+1]-subUk_d[ty+1][tx+1]),2)+pow((subUk_d[ty+1][tx+2]-subUk_d
                    [ty+1][tx+1]),2)+pow(Epsilon,2)) + (subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1])*rsqrt(pow((
                    subUk_d[ty+1][tx+1]-subUk_d[ty][tx+1]),2)+pow((subUk_d[ty][tx+2]-subUk_d[ty][tx+1])
                    ,2)+pow(Epsilon,2))+ (subUk_d[ty+1][tx+1]-subUk_d[ty+1][tx])*rsqrt(pow((subUk_d[ty
                    +2][tx]-subUk_d[ty+1][tx]),2)+pow((subUk_d[ty+1][tx+1]-subUk_d[ty+1][tx]),2)+pow(
                    Epsilon,2));
            }
348
        GradPsixy[begin + lIM*ty + tx] = Lambda*CGrad[begin + lIM*ty + tx].x/(lIM*hIM) + gradPsi;

    }

353 //
    ------------------------------------------------------------------------

    // Fonction permettant de verifier le deroulement des appels aux fonctions CUDA

    extern "C" void Check_CUDA_error(const char *msg)
    {
358     cudaError_t err = cudaGetLastError();
        if( cudaSuccess != err )
```

```
        {
                fprintf( stderr , "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err ) );
                exit (EXIT_FAILURE);
363     }
}
//
    ------------------------------------------------------------------------

// Fonction pour calculer le conjugue de FH
__global__ void Conjuguee(cufftDoubleComplex* aconj, cufftDoubleComplex* Conj, int lIM, int hIM)
368 {
        // Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
373     int ty = threadIdx.y;

        // Index des deux sous matrices utilisees par le block
        int i = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        Conj[i+lIM*ty+tx].x = aconj[i+lIM*ty+tx].x;
378     Conj[i+lIM*ty+tx].y = -1.0 * aconj[i+lIM*ty+tx].y;
}
//
    ------------------------------------------------------------------------

// Fonction pour calculer la premiere partie du gradient
__global__ void CalculTransF(cufftDoubleComplex* FH, cufftDoubleComplex* FUo_d, cufftDoubleComplex*
    FUk_d, cufftDoubleComplex* GradF, int lIM, int hIM)
383 {
        // Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
388     int ty = threadIdx.y;

        // Index des deux sous matrices utilisees par le block
        int i = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        //GradF(i)=FHconj(i)*FH(i)*FUk_d(i)-FHconj(i)*FUo_d(i);
393     GradF[i+lIM*ty+tx].x=(pow(FH[i+lIM*ty+tx].x,2)+pow(FH[i+lIM*ty+tx].y,2))*FUk_d[i+lIM*
            ty+tx].x-(FH[i+lIM*ty+tx].x*FUo_d[i+lIM*ty+tx].x+FH[i+lIM*ty+tx].y*FUo_d[i+lIM*
            ty+tx].y);
        GradF[i+lIM*ty+tx].y=(pow(FH[i+lIM*ty+tx].x,2)+pow(FH[i+lIM*ty+tx].y,2))*FUk_d[i+lIM*
            ty+tx].y-(FH[i+lIM*ty+tx].x*FUo_d[i+lIM*ty+tx].y-FH[i+lIM*ty+tx].y*FUo_d[i+lIM*
            ty+tx].x);
}
//
    ------------------------------------------------------------------------

//Fonction pour convertir des reels en complexes
398 __global__ void realtocomplex(double *Aconvertir, cufftDoubleComplex * Complexe, int lIM, int hIM)
{
        // Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
403     int tx = threadIdx.x;
```

```
        int ty = threadIdx.y;

        // Index des deux sous matrices utilisees par le block
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        Complexe[index+ lIM*ty + tx ].x=Aconvertir[index+ lIM*ty + tx ];
        Complexe[index+ lIM*ty + tx ].y=0.f;

}
//
    ------------------------------------------------------------------------

//Fonction qui convertit les complexes en reels
__global__ void complextoreal(cufftDoubleComplex *Aconvertir, double * Real, int lIM, int hIM)
{
          // Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

          // Index
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        Real[index + lIM*ty + tx]=Aconvertir[index + lIM*ty + tx ].x;
}
//
    ------------------------------------------------------------------------

//Fonction qui calcul le y de la descente de gradient amelioree
__global__ void CalculY (cufftDoubleComplex * CUk_d, double * Uo, double A, double a, double * g,
    cufftDoubleComplex * Cy, int lIM)
{

//A et a : variables de la boucle principale
// Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

       // Index
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        Cy[index + lIM*ty + tx ].x=(CUk_d[index + lIM*ty + tx ].x*A+a*(Uo[index + lIM*ty + tx]-g[index +
            lIM*ty + tx ]) )/(A+a);

}
//
    ------------------------------------------------------------------------

//Fonction qui calcul le nouveau x de la descente de gradient amelioree
__global__ void Calculx (cufftDoubleComplex * CUk_d, cufftDoubleComplex * Cy, double * Grady, double
    Lipschitz, int lIM)
{

//A et a : variables de la boucle principale
// Configuration
```

```
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

       // Index
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        CUk_d[index + lIM*ty + tx ].x=Cy[index + lIM*ty + tx ].x-Grady[index + lIM*ty + tx ]/ Lipschitz ;

}
//
    ------------------------------------------------------------------------

//Fonction qui calcul le nouveau g de la descente de gradient amelioree
__global__ void Calculg (double * g, double * Gradx, double a, int lIM)
{

// Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

       // Index
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        g[index + lIM*ty + tx]=g[index + lIM*ty + tx]+Gradx[index + lIM*ty + tx]*a;

}//
    ------------------------------------------------------------------------

// Initialisation de g
__global__ void Initg (double * g, int lIM)
{

// Configuration
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

       // Index
        int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        g[index + lIM*ty + tx ]=0.0;

}
//
    ------------------------------------------------------------------------

//Fonction d'affichage pour debug
__global__ void putinUk_d (cufftDoubleComplex* ATester, double * A_Afficher, int lIM)
{

// Configuration
            int bx = blockIdx.x;
            int by = blockIdx.y;
```

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

            // Index
503         int index = lIM * BLOCK_SIZE * by + BLOCK_SIZE * bx;
            A_Afficher[index + lIM*ty + tx]=ATester[index + lIM*ty + tx].x;

    }
    //
```

7.2 Compléments CUDA

7.2.1 *Computecapabilities*

7.2.1.a 1.0

- Le nombre maximal de threads par block est 512
- La taille d'un warp est 32
- Le nombre de registres par multiprocesseur est 8192
- La taille de la mémoire partagée est de 16KB par multiprocesseur
- La taille de la mémoire constante est de 64KB
- Le nombre maximal de blocks actifs est de 8 par multiprocesseur
- Le nombre maximal de warps actifs est de 24 par multiprocesseur
- Le nombre maximal de threads actifs est de 768 par multiprocesseur
- Les tailles maximales des composantes x, y et z d'un block sont respectivement 512, 512 et 64
- Les tailles maximales de chacune des composantes de la grille sont 65535

7.2.1.b 1.1

En plus des spécifications ci-dessus :
- Les fonctions atomiques qui opèrent sur des mots de 32 bits en mémoire globale sont supportées

7.2.1.c 1.2

En plus des spécifications ci-dessus :
- Les fonctions atomiques qui opèrent sur des mots de 64 bits en mémoire globale, et celles qui opèrent sur des variables (mots de 32 bits) en mémoire partagée sont supportées
- Les fonctions warp de décision sont supportées
- Le nombre de registres par multiprocesseur est de 16384
- Le nombre maximal de warps actifs par multiprocesseur est de 32
- Le nombre maximal de threads actifs par multiprocesseur est de 1024

7.2.1.d 1.3

En plus des spécifications ci-dessus :
- Les floats double précision sont supportés

7.2.2 Fonctions atomiques arithmétiques

7.2.2.a atomicAdd()

Listing 3 – atomicAdd().cu

```
1 int atomicAdd(int* adresse, int val);
  unsigned int atomicAdd(unsigned int* adresse, unsigned int val);
3 unsigned long long int atomicAdd(unsigned long long int* adresse, unsigned long long int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 ou 64 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val+val**). La fonction retourne **old_val**.

Les mots de 64 bits ne sont supportés que pour la mémoire globale.

7.2.2.b atomicSub()

Listing 4 – atomicSub().cu

```
1 int atomicSub(int* adresse, int val);
2 unsigned int atomicSub(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val-val**). La fonction retourne **old_val**.

7.2.2.c atomicExch()

Listing 5 – atomicExch().cu

```
1 int atomicExch(int* adresse, int val);
  unsigned int atomicExch(unsigned int* adresse, unsigned int val);
3 unsigned long long int atomicExch(unsigned long long int* adresse,unsigned long long int val);
  float atomicExch(float* adresse, float val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 ou 64 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par celle de **val**. La fonction retourne **old_val**.

Les mots de 64 bits ne sont supportés que pour la mémoire globale.

7.2.2.d atomicMin()

Listing 6 – atomicMin().cu

```
1 int atomicMin(int* adresse, int val);
  unsigned int atomicMin(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le minimum de **old_val** et **val**. La fonction retourne **old_val**.

7.2.2.e atomicMax()

Listing 7 – atomicMax().cu

```
1 int atomicMax(int* adresse, int val);
  unsigned int atomicMax(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le maximum de **old_val** et **val**. La fonction retourne **old_val**.

7.2.2.f atomicInc()

Listing 8 – atomicInc().cu

```
1 unsigned int atomicInc(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de ((**old_val >= val**) **? 0 : (old_val+1)**). La fonction retourne **old_val**.

7.2.2.g atomicDec()

Listing 9 – atomicDec().cu

```
1 unsigned int atomicDec(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de ((**old_val == 0 || old_val > val**) **? val : (old_val-1)**). La fonction retourne **old_val**.

7.2.2.h atomicCas()

Listing 10 – atomicCas().cu

```
1 int atomicCAS(int* adresse, int compare, int val);
  unsigned int atomicCAS(unsigned int* adresse, unsigned int compare, unsigned int [.....] val);
  unsigned long long int atomicCAS(unsigned long long int* adresse, unsigned long [.....] long int
      compare, unsigned long long int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 ou 64 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val == compare ? val : old_val**). La fonction retourne **old_val**.

Les mots de 64 bits ne sont supportés que pour la mémoire globale.

7.2.3 Fonctions atomiques bit-à-bit

7.2.3.a atomicAnd()

Listing 11 – atomicAnd().cu

```
1 int atomicAnd(int* adresse, int val);
2 unsigned int atomicAnd(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val & val**). La fonction retourne **old_val**.

7.2.3.b atomicOr()

Listing 12 – atomicOr().cu

```
1 int atomicOr(int* adresse, int val);
  unsigned int atomicOr(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val | val**). La fonction retourne **old_val**.

7.2.3.c atomicXor()

Listing 13 – atomicXor().cu

```
1 int atomicXor(int* adresse, int val);
  unsigned int atomicXor(unsigned int* adresse, unsigned int val);
```

Cette fonction lit un mot (nommé **old_val**) de 32 bits situé à l'adresse **adresse** en mémoire globale ou partagée, et remplace sa valeur par le résultat de (**old_val ^ val**). La fonction retourne **old_val**.

7.2.4 Mode d'accès aux données

7.2.4.a Mémoire globale

On notera sur le schéma :
- A gauche, adressage résultant en une seule transaction mémoire
- A droite, adressage divergent, résultant en une seule transaction

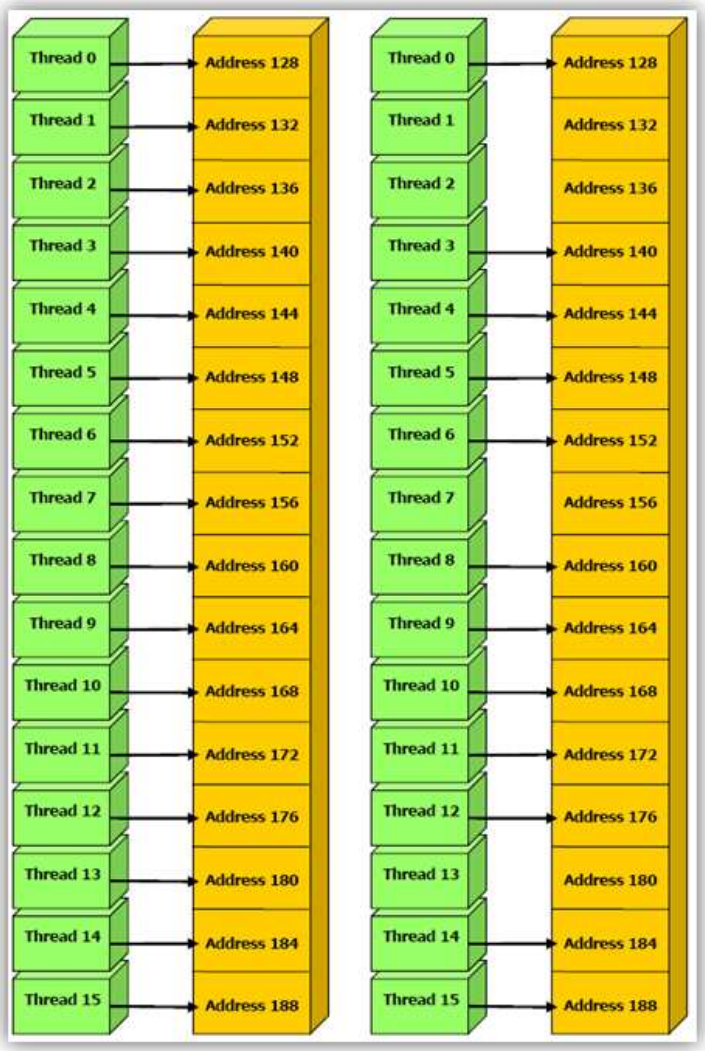

FIG. 19 – Exemples d'adressages amalgamés à la mémoire globale

On notera sur le schéma :
- A gauche, adressage aléatoire dans un segment de 64B, résultant en une seule transaction mémoire
- Au centre, adressage mémoire mal aligné, résultant en une seule transaction mémoire
- A droite, adressage mémoire mal aligné, résultant en deux transactions mémoire

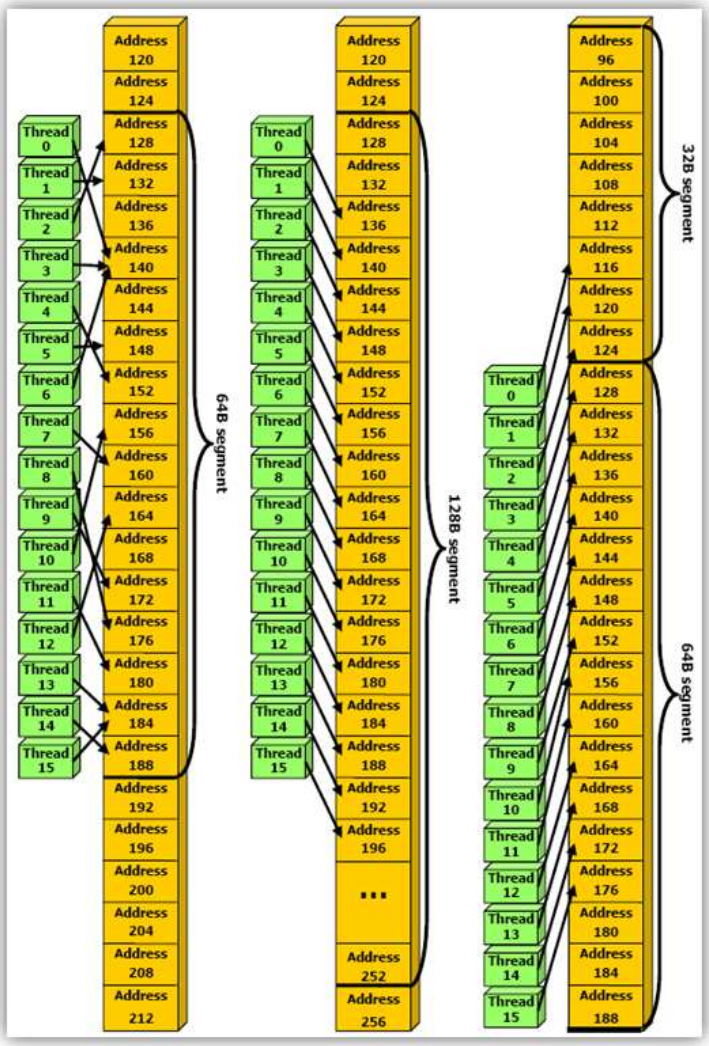

FIG. 20 – Exemples d'adressages à la mémoire globale

7.2.4.b Mémoire partagée

On notera sur le schéma :
- A gauche, adressage linéaire avec un saut d'un mot de 32 bits
- A droite, adressage aléatoire

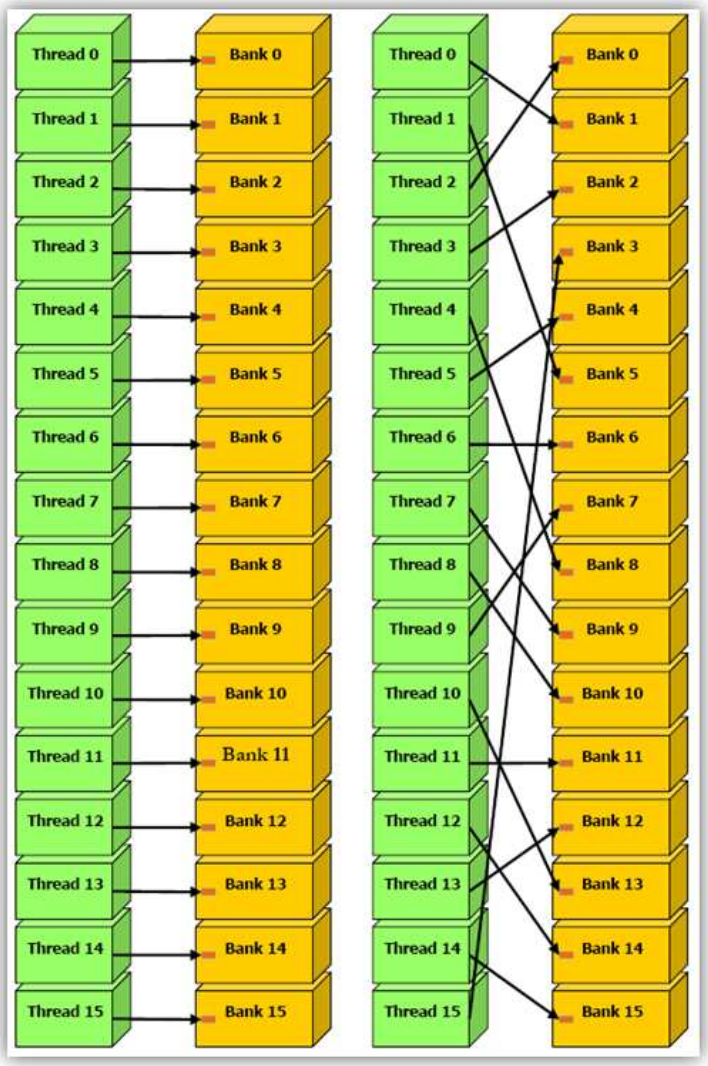

FIG. 21 – Exemples d'adressages à la mémoire partagée sans conflit de banks

On notera sur le schéma un adressage linéaire avec un saut d'un mot de 32 bits.

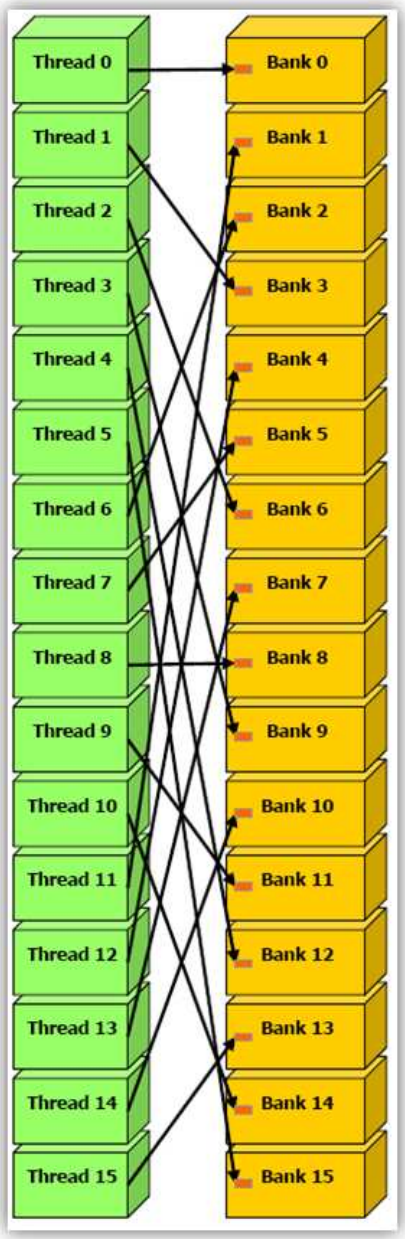

FIG. 22 – Exemple d'adressage à la mémoire partagée sans conflit de banks

On notera sur le schéma :
- A gauche, adressage linéaire avec un saut de deux mots de 32 bits causant deux chemins de conflits de banks
- A droite, adressage linéaire avec un saut de huit mots de 32 bits causant huit chemins de conflits de banks

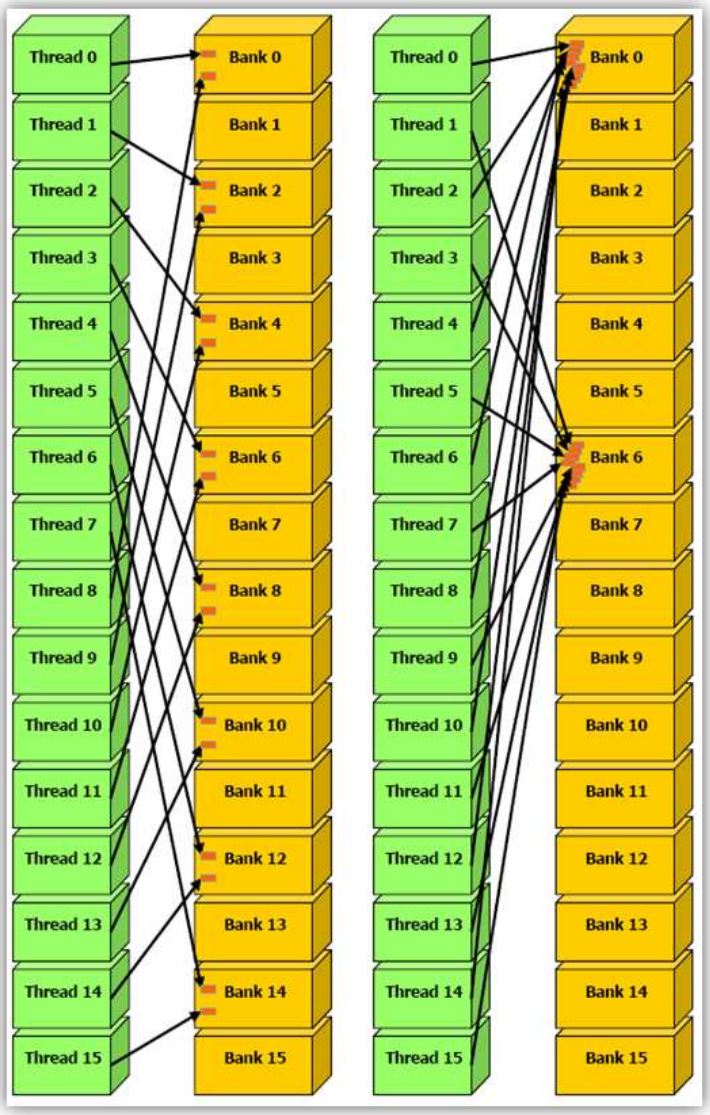

FIG. 23 – Exemples d'adressages à la mémoire partagée avec conflits de banks

On notera sur le schéma l'absence de conflit de banks

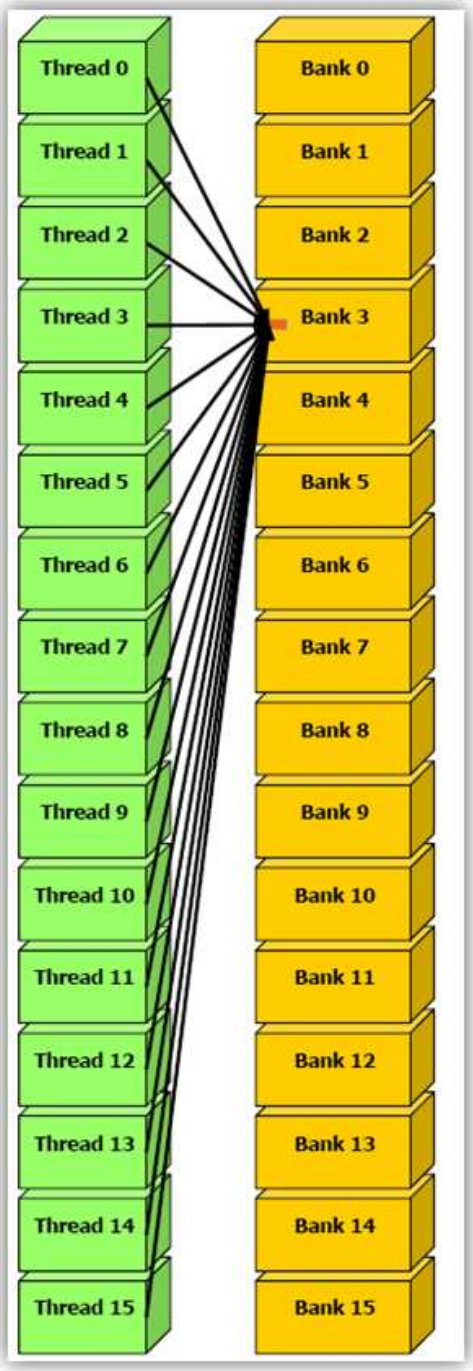

FIG. 24 – Exemple d'adressage à la mémoire partagée avec " broadcast "

Références

[1] Antonin CHAMBOLLE, *An algorithm for Total Variation Minimization and Applications*. Journal of Mathematical Imaging and Vision 20 :89-97, 2004.

[2] Yurii NESTEROV *Introductory Lectures On Convex Optimization*. 2004.

[3] Leonid I. RUDIN, Stanley OSHER et Emad FATEMI *Non Linear Total Variation Based Noise Removal Algorithms*. Physica D.

[4] Khalid JALALZAI, Antonin CHAMBOLLE. *Restauration d'images floutées et bruitées par une variante originale de la variation totale*.

[5] `http://www.nvidia.fr/object/cuda_what_is_fr.html`