Nom : YAOKELI LIKOKE

Prénom : JEAN

# Stage de fin d'année
# Cycle Ingénieur 4ème année

Département : ELECTRONIQUE ET SYSTEMES EMBARQUES

Tuteur école : CEDRIC KOENIGUER
cedric.koeniguer@upsud.fr, 0169338609

**Optimisation du transfert de données entre un processeur central (CPU) et un accélérateur de calcul (GPU)**

**Société** : Fédération de Recherche LUmière MATière (LUMAT)
**Adresse** : Bât 210, Université Paris Sud, 91405 Orsay Cedex
**Dates du stage** : 29/05 au 31/07 et 20/08 au 07/09/2012
**Tuteur de l'entreprise** : Philippe Dos Santos
**Téléphone** : 0169158255

# Remerciements

Je remercie Philippe Dos Santos ainsi que Georges Raseev pour le stage sur la grappe massivement parallèle de calcul scientifique (GMPCS) de la fédération LUmière-MATière (LUMAT). Leur disponibilité, soutien et conseils m'ont permis d'accomplir mon travail dans les meilleures conditions.

Je tiens à remercier tous les chercheurs, enseignants-chercheurs, ingénieurs, techniciens et administratifs de la fédération LUMAT pour leur accueil chaleureux.

Je tiens aussi à remercier Cédric Koeniguer d'avoir été toujours là pour nous encourager durant nos recherches de stage et de son soutien pendant ce dernier, sans oublier Hervé Mathias pour son assistance tout au long de mon stage.

## SOMMAIRE

**Remerciements**

**Introduction**

1. **Cadre de travail : la fédération de recherche LUmière-MATière (LUMAT)** ................5
   1.1 Les laboratoires …...........................................................................................5
   1.2 Les plates-formes …..........................................................................................6
   1.3 La mise en commun des moyens de calcul …....................................................6
      1.3.1 Présentation de la GMPCS …....................................................................6
      1.3.2 Architecture matérielle et logicielle générale .............................................7

2. **Présentation et comparaison des unités de calcul CPU et GPU** .................................9
   2.1 Lexique …...........................................................................................................9
   2.2 Processeurs centraux et accélérateurs de calcul................................................11
      2.2.1 L'unité centrale de traitement(CPU) …......................................................11
      2.2.2 Les processeurs graphiques GPU de Nvidia …...........................................11
      2.2.3 Les accélérateurs à base de CPU …............................................................13

3. **Optimisation du transfert de données CPU-GPU dans le cas de la transformée de Fourier rapide (FFT)** …..........................................................................................14
   3.1 Algorithme de calcul de la FFT.........................................................................14
   3.2 Calcul FFT sur GPU sans transfert de données …...........................................16
   3.3 Transfert de données entre le CPU et le GPU …..............................................17
      3.3.1 Type d'accès à la mémoire …..................................................................17
      3.3.2 Transfert synchrone et asynchrone …......................................................18
   3.4 Cas réel : calcul FFT sur GPU avec transfert de données........................................22

**Conclusion** ...........................................................................................................23

**Références bibliographiques et numériques** .......................................................24
**Annexes** ...............................................................................................................26

## Introduction

La Grappe Massivement Parallèle de Calcul Scientifique est une des plate-formes de la fédération Lumière-Matière (LUMAT). Elle est composée d'un nœud maître connecté au réseau internet et de 25 nœuds dédiés au calcul scientifique dont chacun est équipe de des deux processeurs à 4-6 cœurs.

Deux nœuds de calcul associés sont à des accélérateurs de type Graphics Processing Unit (GPU). L'accélérateur de calcul GPU n'est pas une unité de calcul indépendante car il ne dispose pas de système d'exploitation.

Les cartes graphiques récentes possèdent toutes un GPU permettant le traitement rapides des images 2D et 3D visualisées sur les écrans d'ordinateurs. La particularité des GPU est liée au fait qu'il possèdent plusieurs processeurs ayant une architecture beaucoup plus simple qu'un processeurs d'ordinateur mais permettant le traitement massivement parallèle des données graphiques.

Depuis 2008, les GPU associés aux Central Processing Unit (CPU) sont utilisés pour le calcul numérique. Cette conversion numérique nécessite de reprogrammer le software des GPU,étant donné que ces derniers n'acceptent que des instructions spécifiques de type graphique.

Les étapes de calcul utilisant un accélérateur graphique sont : transfert de données CPU vers GPU, calcul sur l'accélérateur graphique GPU, transfert GPU vers CPU des résultats. L'analyse faite pendant ce travail du temps nécessaire à la réalisation de chacune des étapes citées montre que le goulot d'étranglement se situe au niveau du transfert de données CPU vers GPU et GPU vers CPU.

Ce stage traite de la stratégie en général pour optimiser le transfert de données entre l'espace mémoire du CPU et celui des GPU associés pour optimiser globalement l'utilisation du GPU.

# 1. Cadre de travail : la fédération de recherche LUmière-MATière (LUMAT)

## 1.1 Les laboratoires

La fédération LUmière MATière (LUMAT) est une fédération de recherche, regroupant quatre laboratoires de recherche sur le campus d'Orsay : le Laboratoire Aimé Cotton (LAC, UPR 3321), le Laboratoire des Gaz et des Plasmas (LPGP, UMR 8578), le Laboratoire Charles Fabry (LCF, UMR 8501) et l'Institut des Sciences Moléculaires d'Orsay (ISMO, UMR 8214).

Cette fédération est composée d'environ 320 permanents rattachés au Centre National de la Recherche Scientifique (CNRS), à l'Université Paris Sud et à l'Institut d'Optique Graduate School.
Les 320 permanents de la fédération sont constitués de :

- 115 chercheurs
- 75 enseignants-chercheurs
- 130 ingénieurs, techniciens et administratifs

Le but de la fédération LUMAT est de mutualiser les plates-formes et le savoir-faire entre les différentes unités de recherche qui la constituent, en passant par le développement des projets scientifiques communs.

La fédération LUMAT joue également un rôle important dans l'animation scientifique entre les différentes équipes de recherche qui la composent grâce au financement des projets de recherche.
L'organigramme de la fédération LUMAT se présente de la manière suivante :

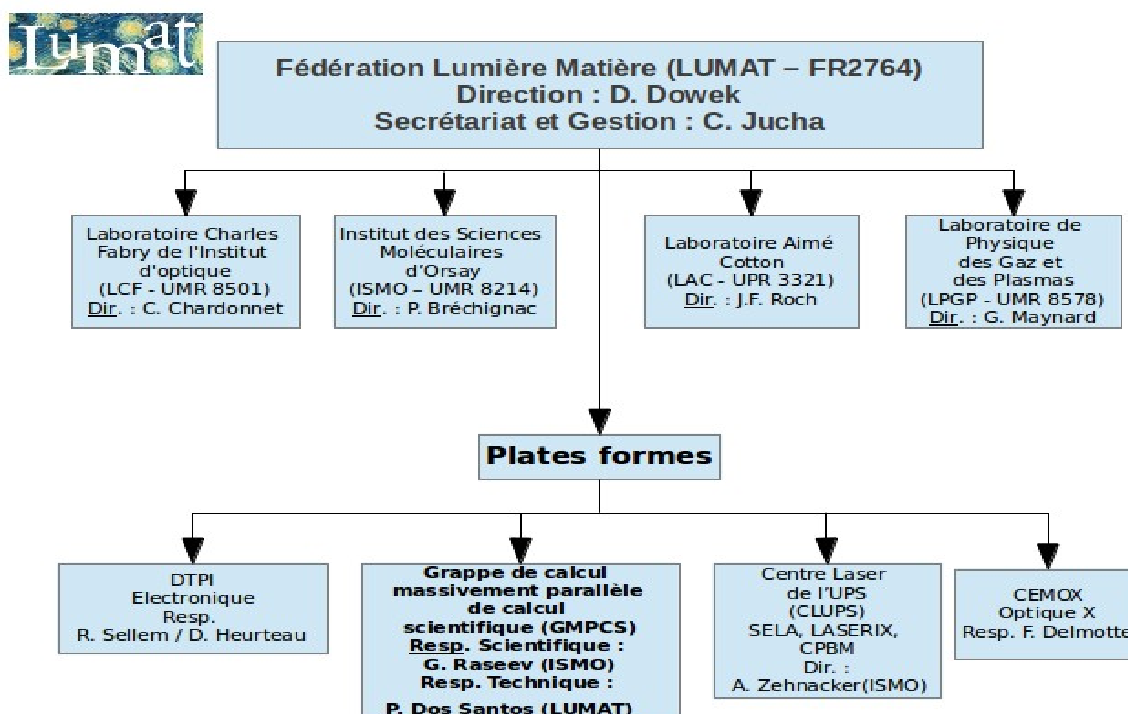


Figure 1: L'organigramme de la fédération LUmière-MATière (LUMAT)

### 1.2 Les plates-formes

Plusieurs plates-formes instrumentales ayant des interfaces avec d'autres branches scientifiques sont fédérées par le LUMAT :

- le Centre Laser de l'université Paris Sud (CLUPS)
- le Centre de Photonique Biomédicale (CPBM) et le LASERIX
- la Grappe Massivement Parallèle de Calcul Scientifique (GMPCS)
- la Centrale d'Élaboration et de Métrologie des Optiques X-UV (CEMOX)
- la plate-forme Détection-Temps-Position-Image (DTPI)

### 1.3 La mise en commun des moyens de calcul

La mise en commun des moyens de calcul des laboratoires de la fédération LUMAT s'est matérialisée par l'acquisition en 2008 de la Grappe Massivement Parallèle de Calcul Scientifique (GMPCS) offrant aux chercheurs des moyens de calcul plus performants que ceux disponibles à l'échelle d'un laboratoire.

#### 1.3.1 Présentation de la GMPCS

La GMPCS est un mésocentre, plus précisément un calculateur de taille intermédiaire situé entre la station de travail (Tier 3) et les centres de calcul nationaux (Tier 1) comme le montre la figure 2.

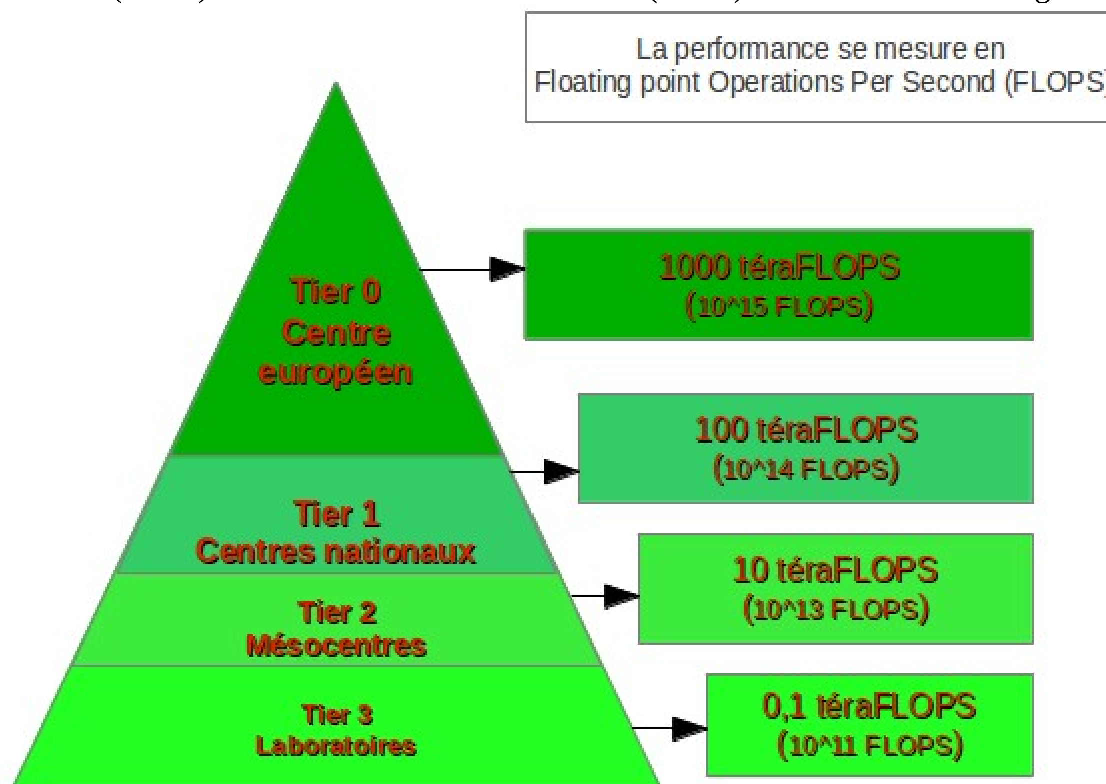


Figure 2: Classement des centres en fonction de la puissance de calcul

### 1.3.2 Architecture matérielle et logicielle générale

Assemblée par la société Transtec, la GMPCS est un ensemble d'unités de calcul (processeurs) regroupés en nœuds. Ces nœuds sont connectés à un réseau dédié au calcul, cette architecture permet à la GMPCS d'effectuer des calculs en parallèle sur plusieurs nœuds et dans certains cas, cela additionne la puissance de calcul des nœuds.

La GMPCS est composée :

- d'un nœud maître
- et de 25 nœuds dédiés au calcul.

Le nœud maître est l'unique point d'entrée à la GMPCS, il joue donc le rôle :

- point d'accès : c'est le seul point d'entrée sur la GMPCS. Il est directement accessible via l'internet par le protocole SSH.
- gestion des jobs : il réserve les ressources sur les nœuds de calcul (nombre de cœurs, quantité mémoire, durée) et distribue les jobs des utilisateurs en veillant à l'équilibre de la charge de la GMPCS via le logiciel SGE.
- gestion des nœuds : il gère le déploiement du système d'exploitation au démarrage des nœuds via le logiciel xCAT.
- suivi de la charge : il permet de suivre la charge de l'ensemble des nœuds via ganglia.
- suppression : il détecte tout défaut hardware et le signale via Nagios
- gestion des données : toutes les données envoyées par les utilisateurs pour le calcul sont tout d'abord stockées sur le nœud maître, puis par la suite accessibles sur les nœuds de calcul au moyen du partage interne de fichiers.

La figure 3 montre l'architecture de la GMPCS :

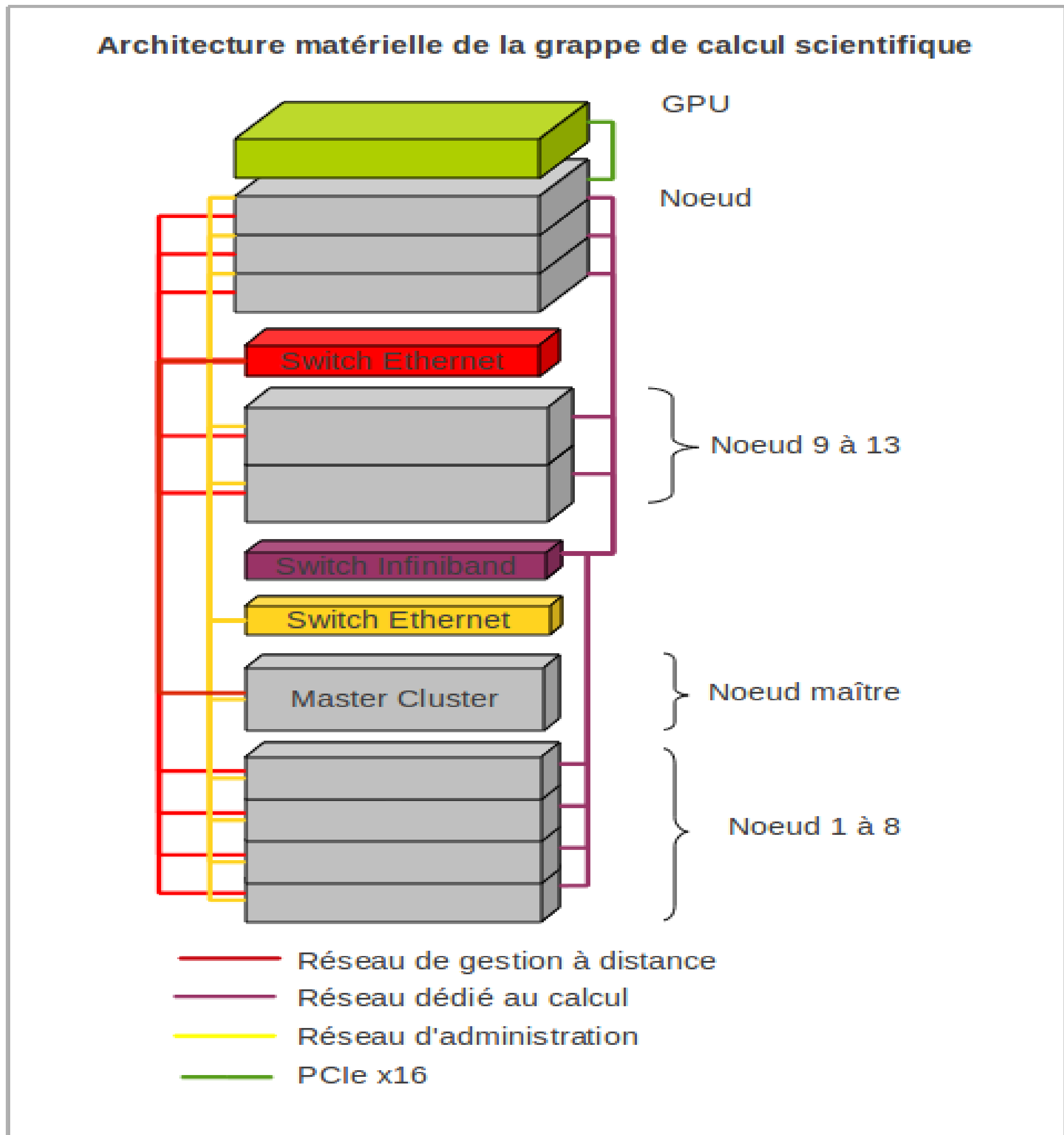


Figure 3: Le schéma de la GMPCS

Les 25 nœuds de calcul sont dédiés uniquement aux calculs.

Ces nœuds sont connectés à 3 réseaux avec des fonctionnalités complètement différentes :

- le réseau d'administration utilisant le protocole Ethernet, est un réseau utilisé pour le partage des dossiers entre les utilisateurs et également pour le transfert des codes sur les nœuds.
- Le réseau de gestion à distance permet de gérer l'arrêt et le démarrage des noeuds.
- Le réseau infiniband (réseau rapide dédié aux calculs), est un réseau principalement utilisé pour les calculs parallèles.

## 2. Présentation et comparaison des unités de calcul CPU et GPU

La figure ci-dessous montre les différentes étapes associées à l'utilisation du GPU. Pour effectuer un calcul sur l'accélérateur GPU :

1. les données sont transférées de la mémoire du CPU vers la mémoire du GPU
2. le GPU traite les données
3. les données sont transférées de la mémoire du GPU vers la mémoire du CPU

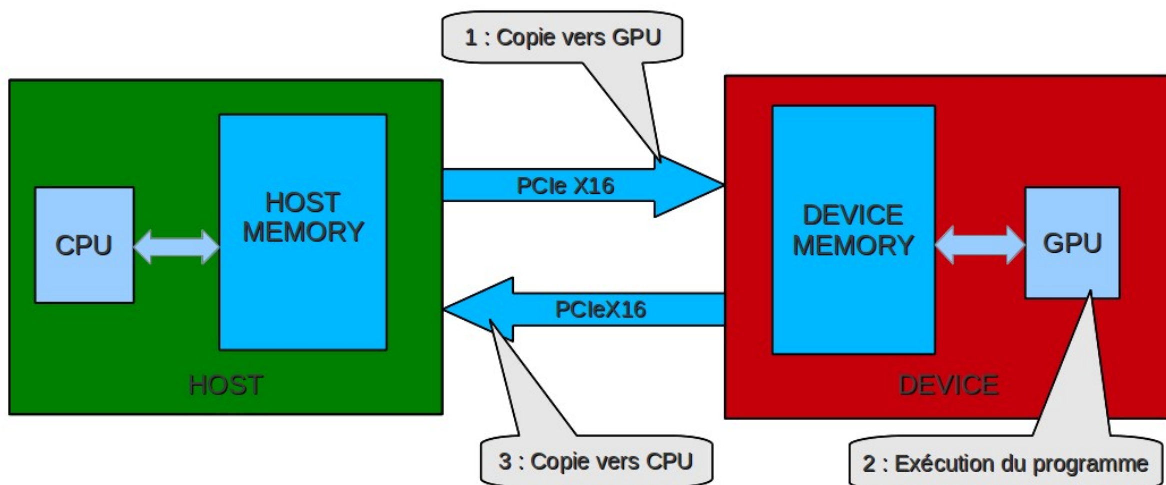


Figure 4: Étapes associées à l'utilisation du GPU

Pour optimiser l'utilisation de l'accélérateur GPU, ces différentes étapes seront analysées dans ce travail.

### 2.1 Lexique

Avant de commencer, je tiens à donner quelques définitions :

**Mode de calcul**

- ➔ **Calcul séquentiel** : un calcul en série dont les opérations sont exécutées les unes après les autres. Un cœur de CPU est un exemple d'un processeur à calcul séquentiel.
- ➔ **Calcul parallèle** : un calcul pouvant être scindé en plusieurs opérations distinctes exécutées en parallèles. Le GPU est un exemple de processeur à calcul parallèle.

**Architectures des processeurs**

- ➔ **SISD (**Single Instruction Single Data**) :** une seule instruction est appliquée sur une seule donnée pour produire un seul résultat.

- ➔ **SIMD (**Single Instruction on Multiple Data **) :** une même instruction est appliquée simultanément sur plusieurs données pour produire plusieurs résultats.
- ➔ **MIMD (**Multiple Instruction Multiple Data**) :** plusieurs instructions sont appliquées sur plusieurs données pour produire plusieurs résultats.

**Host (Hôte)**

- ➔ **CPU** (Central Processing Unit): unité centrale de traitement de l'ordinateur, le système d'exploitation permet d'ordonnancer les tâches que le CPU effectue d'une manière séquentielle.
- ➔ **Mémoire CPU (ordinary memory):** mémoire vive nécessaire au CPU pour l'exécution des programmes.
- ➔ **Mémoire DMA (Direct Memory Access) ou pinned memory:** espace mémoire permettant au GPU d'avoir un accès direct aux données sans transiter par le CPU.

**Accélérateur ou Device (Périphérique) :**

- ➔ **GPU:** en dehors de l'unité de calcul principale (CPU), le GPU est considéré comme un périphérique. C'est un processeur spécialisé sans système d'exploitation. Il est géré par le CPU (HOST) et possède une architecture massivement parallèle (plus de 400 cœurs). Le GPU est un exemple d'accélérateur de calcul.
- ➔ **Kernel:** programme sur le GPU qui permet de transférer les données par blocs, d'exécuter les opérations associées sur ce dernier et écrire le résultat dans la mémoire du GPU.
- ➔ **Stream:** séquence des opérations sur le GPU qui s'exécutent les unes après les autres dans un ordre spécifique.
- ➔ **Thread:** processus d'un kernel traitant d'une manière séquentielle un bloc de données.
- ➔ **Warp:** groupe de 32 threads s'exécutant d'une manière parallèle.
- ➔ **Cuda (Compute Unified Device Architecture):** langage de programmation développé par NVIDIA permettant de programmer un GPU. Il est basé sur le langage de programmation C complété d'un jeu d'instructions supplémentaire dédié au GPU.

**Transferts :**

- ➔ **Transfert synchrone** : transfert de données bloquant, ne permettant pas l'exécution simultanément de plusieurs opérations entre le CPU et le GPU. Il effectue une seule opération à la fois, c'est à dire, il copie les données du CPU vers GPU.
- ➔ **Transfert asynchrone** : transfert de données non bloquant qui permet d'effectuer des opérations en parallèle. Ce mode de transfert peut utiliser les streams pour empiler une suite d'opérations jusqu'à atteindre la fin de l'exécution du programme.

## 2.2 Processeurs centraux et accélérateurs de calcul

### 2.2.1 L'unité centrale de traitement (Central Processing Unit : CPU)

L'unité centrale de traitement est l'élément de base servant à exécuter les instructions d'un programme sur l'ordinateur. Plus précisément le CPU contient des unités de calcul très performantes avec une structure de contrôle permettant d'exécuter d'une manière optimisée une série de tâches séquentielles.

Ces d'unités de calcul fonctionnent en séquentiel en suivant le modèle Single Instruction Single Data (SISD). Dans ce modèle, une seule instruction est appliquée sur une seule donnée pour produire un seul résultat.

Les processeurs modernes disposent de plusieurs CPU appelés cœur. Il est possible de les utiliser suivant le modèle Multiple Instruction Multiple Data (MIMD). Dans ce modèle, chaque cœur traite une donnée différente car chaque cœur est indépendant.

La figure 5 montre l'architecture d'un CPU (HOST) à 6 cœurs :

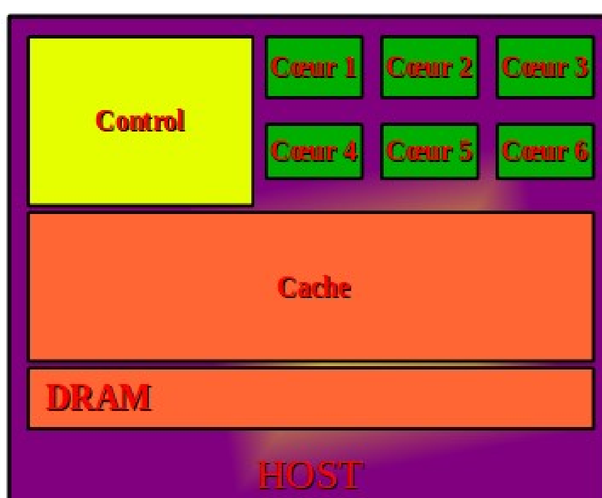


Figure 5 : architecture d'un CPU à 6 cœurs

### 2.2.2 Les processeurs graphiques GPU de Nvidia

Le GPU est un regroupement d'unités de calculs à faible fréquence moins performantes qu'une unité de calcul CPU. Ces unités de calcul fonctionnent en parallèle en suivant le modèle Single Instruction on Multiple Data (SIMD). Dans ce modèle, une même instruction est appliquée simultanément sur plusieurs données pour produire plusieurs résultats. Ce modèle est particulièrement bien adapté au calcul matriciel.

Un modèle de l'architecture de GPU (DEVICE) est présenté sur la figure 6 :

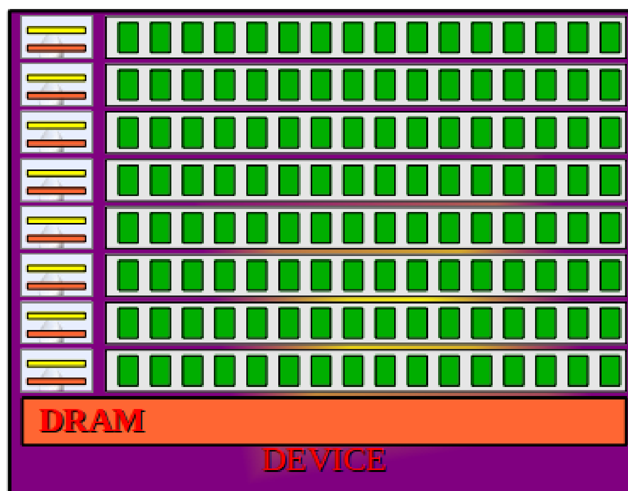


Figure 6 : architecture d'un GPU

Le tableau ci-dessous présente les différentes évolutions des processeurs Nvidia pour le calcul.

| | Tesla C870 | Tesla C1060 | Tesla C2050 | Tesla C2090 | Tesla K10a |
|---|---|---|---|---|---|
| **Architecture** | Tesla 8-series | Tesla 10-series | Tesla 20-series | Tesla 20-series | Tesla K10-series (2 GPU) |
| **Nombre de cœurs** | 128 | 240 | 448 | 512 | 2 x 1536 |
| **Fréquence** | 1.35 GHz | 1.3 GHz | 1.15 GHz | 1.3 GHz | N/A |
| **Performance :**<br>**- single**<br>**- double** | 345.6 Gflops | 933 Gflops<br>78 Gflops | 1030 Gflops<br>515 Gflops | 1332.2 Gflops<br>666.1 Gflops | 2 x 2290 Gflops<br>2 x 950 Gflops |
| **Mémoire** | 1536 MB | 4 GB | 3 GB | 6 GB | 2 x 4 GB |
| **Bande passante mémoire** | 76.8 Gbps<br>GDDR3 | 102 GB/s<br>GDDR3 | 144 GB/s<br>GDDR5 | 177 GB/s<br>GDDR5 | 2 x 160 GB/s<br>GDDR5 |
| **Système E/S** | PCIe x16<br>Gen1 | PCIe x16<br>Gen2 | PCIe x16 Gen2 | PCIe x16 Gen2 | N/A |

Figure 7 : Évolution des processeur GPU Nvidia pour le calcul

L'accélérateur Nvidia S2050 est le modèle de processeur GPU dont dispose la plate-forme de la fédération LUMAT et sur lequel j'ai pu effectuer l'ensemble de mon travail de stage.

Les caractéristiques de ce modèle d'accélérateur sont les suivantes :

- 448 x 4 unités de calcul

- une précision de la virgule flottante à 64 bits
- 3 x 4 Go de mémoire principale

La figure ci-dessous présente une brève description de l'architecture d'un accélérateur S2050 présent sur la GMPCS

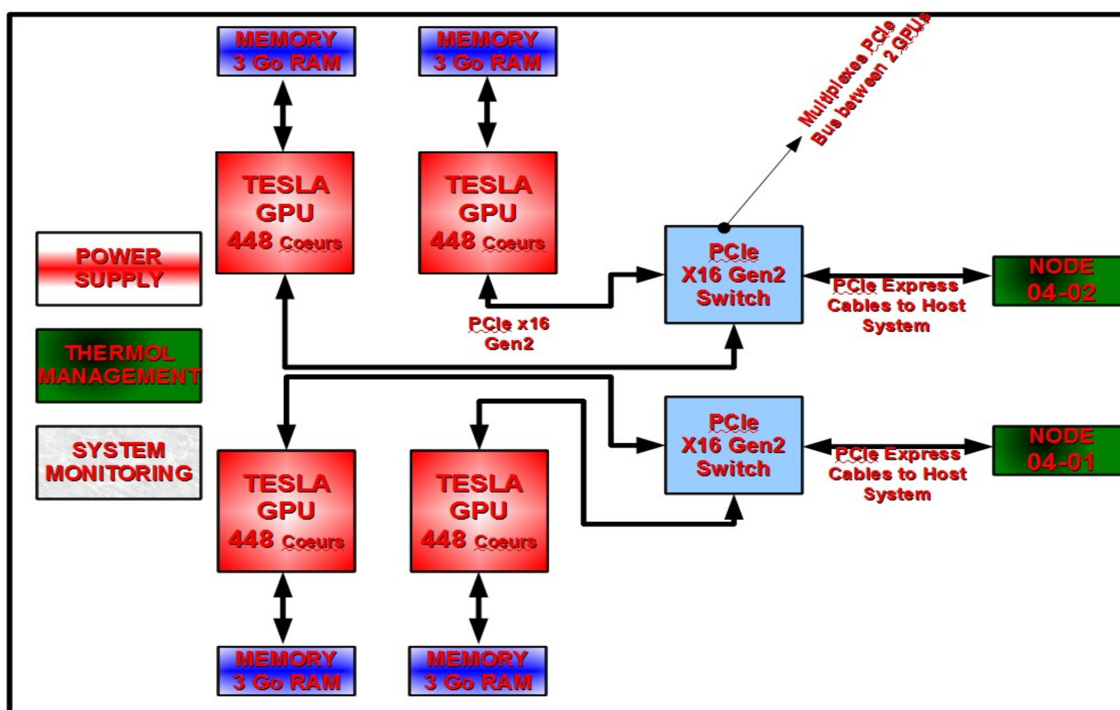


Figure 8 : Tesla S2050 System Architect

Lancé au mois de mars 2012, l'architecture Kepler est l'architecture graphique de nouvelle génération élaborée par NVIDIA. Ce processeur offre beaucoup plus d'amélioration par rapport à l'ancienne génération (FERMI), plus particulièrement en matière d'efficacité énergétique et de performance de calcul. Le nombre d'unités de calcul sur l'accélérateur GPU utilisant cette nouvelle technologie est passé de 448 unités à 1536 unités, rendant ce dernier trois fois plus performant que la version précédente.

Attendus en 2014, la nouvelle architecture Maxwell utilisera une finesse de gravure approchant les 20 nanomètres sachant que Kepler et Fermi utilisent chacun une finesse de gravure de 28 nm et de 40 nm, ceci permettra encore de monter en puissance et de gagner en performance.

2.2.3 Les accélérateurs à base de CPU : le Many Integrated Cores (MIC) d'Intel

Le MIC est une architecture massivement parallèle développée par Intel, intégrant sur une puce de 50 à 100 CPU d'Intel type x86. Les informations sur cette architecture pas encore commerciale sont rares. Il semble que le bus PCIe de transfert de données CPU vers GPU soit étendu permettant de réduire le goulot d'étranglement associé à ce transfert. L'intérêt principal de l'accélérateur à base de CPU est que les programmes existant pour CPU nécessitent une recompilation du code et non une réécriture comme dans le cas du GPU.

## 3. Optimisation du transfert de données CPU-GPU dans le cas de la transformée de Fourier rapide (FFT)

Le but de ce travail consiste à mettre en place le transfert asynchrone de données entre les mémoires du Host et du Device dans l'environnement de la Grappe Massivement Parallèle de Calcul Scientifique.

Après la réalisation du transfert asynchrone de données, le calcul de la FFT pour différentes tailles de données est effectué sur l'accélérateur de calcul GPU. Ceci permet de comparer les performances entre les opérations utilisant le transfert synchrone, celles utilisant le transfert asynchrone et le CPU à 6 cœurs d'un Host (Xeon 5650).

Plusieurs étapes ont été franchies avant la mise en œuvre du transfert de données asynchrone entre le Host et le Device. Et chacune d'elles ont été réalisées les unes à la suite des autres avec comme but final l'optimisation de transfert de données, réduisant ainsi d'une façon considérable le temps global de calcul avec l'accélérateur GPU.

Les 3 grandes étapes aboutissant à la mise en œuvre et à l'optimisation du transfert de données asynchrone ont été organisées comme suit :

- première étape : nous avons commencé tout d'abord par effectuer une recherche sur le type de mémoire à utiliser
- deuxième étape : transfert de données asynchrone entre les deux processeurs
- troisième étape : nous optimisons le transfert de données asynchrone en faisant du chevauchement de données.

À ce stade, il est important de rappeler que l'utilisation de l'accélérateur GPU est composé de 3 phases :

1. transfert de donnée depuis l'espace mémoire du Host vers l'accélérateur GPU
2. appel d'une fonction s'exécutant sur le GPU, habituellement appelé « kernel »
3. récupération des résultats (transfert de données de la mémoire de l'accélérateur GPU vers le Host)

### 3.1 Algorithme de calcul de la FFT

Par définition, [18] et [20], la Transformée de Fourier Discrète (TFD) d'un vecteur complexe f = $(f_0, f_1, \ldots, f_{N-1})$ de taille N est un vecteur complexe F = $(F_0, F_1, \ldots, F_{N-1})$ de taille N dont les composantes sont données par la formule :

$$F_k = \sum_{j=0}^{N-1} f_j \mathrm{e}^{-2\pi i k j/N}, \quad k=0,1,\ldots,N-1 \quad \text{avec} \quad i^2=-1 \quad (1)$$

La TFD inverse qui permet de revenir au vecteur f initial à partir de F est donnée par la formule :

$$f_j = \frac{1}{N}\sum_{k=0}^{N-1} F_k e^{2\pi i k j/N}, \quad j=0,1,\ldots,N-1 \quad (2)$$

Appliquer directement la formule de la TFD nécessite de " l'ordre de N² opérations ", noté O(N²), où " opération " signifie une multiplication complexe suivie d'une addition complexe.

En effet, en posant :

$$W \equiv e^{-2\pi i/N} \quad (3)$$

La formule (1) s'écrit :

$$F_k = \sum_{j=0}^{N-1} f_j W^{kj}, \quad k = 0, 1, \ldots, N-1 \quad (4)$$

Ce qui correspond à la multiplication du vecteur f de taille (N, 1) par une matrice de taille (N, N) dont l'élément (k, j) est W à la puissance k x j. Ce produit matriciel nécessite N² multiplications ainsi que N x (N-1) additions.

Il existe un algorithme rapide de calcul de la TFD qui ne nécessite qu'un nombre d'opérations en O(N $\log_2$ (N)) pour un vecteur de taille N = $2^n$ : c'est l'algorithme Fast Fourier Transform (FFT). L'idée de cet algorithme est que pour N pair, il est possible d'écrire la TFD d'un vecteur de taille N comme une somme de deux TFD dont les vecteurs ont la taille N/2. Pour ce faire, il suffit de réécrire la formule (1) de la façon suivante :

$$F_k = \sum_{j=0}^{N-1} f_j \mathrm{e}^{-2\pi i k j/N}, \quad k = 0, 1, \ldots, N-1$$

$$F_k = \sum_{j=0}^{N/2-1} f_{2j} \mathrm{e}^{-2\pi i k (2j)/N} + \sum_{j=0}^{N/2-1} f_{2j+1} \mathrm{e}^{-2\pi i k (2j+1)/N}$$

$$F_k = \sum_{j=0}^{N/2-1} f_{2j} \mathrm{e}^{-2\pi i k j/(N/2)} + \sum_{j=0}^{N/2-1} f_{2j+1} \mathrm{e}^{-2\pi i k j/(N/2)} \mathrm{e}^{-2\pi i k/N} \quad (5)$$

$$F_k = \sum_{j=0}^{N/2-1} f_{2j} \mathrm{e}^{-2\pi i k j/(N/2)} + \sum_{j=0}^{N/2-1} f_{2j+1} \mathrm{e}^{-2\pi i k j/(N/2)} W^k$$

Dans cette formule :

- On note la première somme $F_k^e$ , qui correspond à la Transformée de Fourier du vecteur de taille N/2 constitué des éléments d'indice pair de f,
- Dans la deuxième somme, $W^k$ ne dépend pas de j, il peut être extrait de cette somme,
- On note alors la deuxième somme $F_k^o$ , qui correspond à la Transformée de Fourier du vecteur de taille N/2 constitué des éléments d'indice impair de f.

$$F_k^e = \sum_{j=0}^{N/2-1} f_{2j} \mathrm{e}^{-2\pi i k j/(N/2)}, F_k^o = \sum_{j=0}^{N/2-1} f_{2j+1} \mathrm{e}^{-2\pi i k j/(N/2)}, \quad k = 0, 1, \ldots, N-1 \quad (6)$$

Comme $F_k^e$ et $F_k^o$ sont périodiques, pour obtenir la transformée de Fourier complète sur N éléments, on fait deux cycles, un de 0 à N/2 - 1 et l'autre de N/2 à N – 1 :

$$F_k = F_k^e + W^k F_k^o \quad pour\, k = 0, 1, \ldots, N-1 \quad (7)$$

Si f est de taille $2^n$, cette opération peut-être exécutée de façon récursive jusqu'à des vecteurs de taille 1 dont la TFD est le vecteur lui-même. L'algorithme complet est de type " Divide and Conquer " et s'écrit :

$$F_k = F_k^e + W^k F_k^o$$

$$F_k = (F_k^{ee} + W^k F_k^{eo}) + W^k (F_k^{oe} + W^k F_k^{oo}) \quad (8)$$

$$F_k = ((F_k^{eee} + W^k F_k^{eeo}) + W^k (F_k^{eoe} + W^k F_k^{eoo})) + W^k ((F_k^{oee} + W^k (F_k^{oeo})) + W^k ((F_k^{ooe}) + W^k (F_k^{ooo})))$$

$$F_k = \ldots$$

Jusqu'à des éléments du type $F_k^{eoeeoeo\ldots oee} = f_n$ où les $f_n$ sont des vecteurs de taille 1.

L'équation de la complexité est donnée par la relation :

$$C(N) = 2C(N/2) + K N \quad (9)$$

La solution de cette relation, [18] et [20], est :

$$C(N) = K N \log_2(N) \quad (10)$$

K varie suivant les implémentations de l'algorithme de la FFT et le but est de le rendre aussi petit que possible.

### 3.2 Calcul FFT sur GPU sans transfert de données

La figure 9 présente les accélérations respectives obtenues pour un CPU à 6 cœurs et un GPU sans le transfert de données.

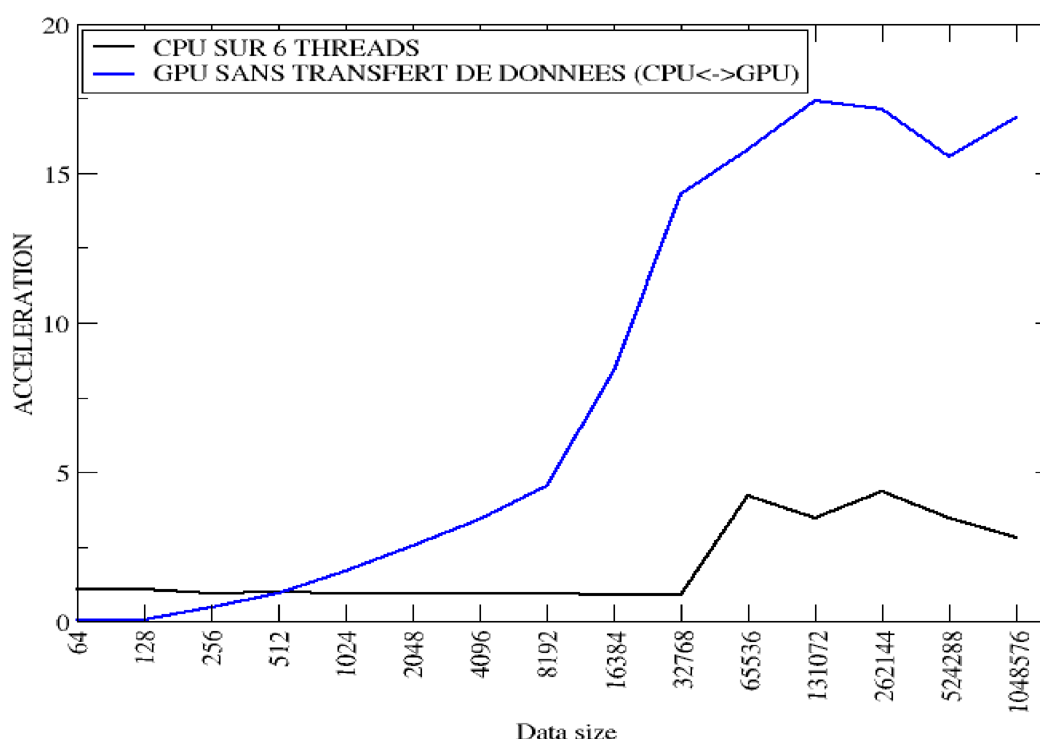


Figure 9 : Accélération brute au GPU comparé au CPU à 6 cœurs
L'accélération est mesurée par rapport à la FFT exécutée sur 1 cœur du CPU

Premièrement, ce graphique montre que les accélérations théoriques augmentent considérablement avec le nombre de points. Notons que le rapport du nombre de processeurs GPU/CPU est de 448/6 ~ 74 donc dans le meilleur des cas un processeur GPU est ~ 74/20 > 3.7 fois plus lent qu'un processeur CPU. Mais globalement le gain est considérable.

Le prochain pas est d'analyser les stratégies les plus efficaces de transfert de données CPU-GPU.

### 3.3 Transfert de données entre le CPU et le GPU

#### 3.3.1 Type d'accès à la mémoire

L'optimisation de la zone mémoire est l'une des étapes la plus importante lors du transfert de données entre le CPU (Host) et l'accélérateur GPU (Device).

En effet, il y existe deux types de fonctions permettant d'allouer préalablement la mémoire :

- «malloc()» alloue la mémoire sur le CPU par conséquent toutes les requêtes de données dont l'accélérateur GPU a besoin doivent d'abord transiter par CPU. Ceci ralentit énormément le transfert de données sur le GPU (Device)
- «cudaHostAlloc()» une fonction CUDA qui permet d'allouer de la mémoire sur le CPU (Host) pour le GPU sans passer par le CPU.

Le type de mémoire qui permet au GPU d'avoir accès directement aux données sans passer par le CPU est appelé «page-locked memory» ou «pinned memory» ou encore «direct access memory (DMA)».

Connaissant l'adresse physique du buffer, l'accélérateur GPU utilise la DMA pour accéder directement aux données dans la mémoire du HOST. Sur le bus PCIe x 16 Gen2, par exemple le pinned memory peut atteindre une vitesse de transfert de 5 GB/s alors que la bande passante maximale de ce dernier est d'environ 8 GB/s.

La figure 10 montre la différence entre le transfert de données utilisant la mémoire réservée avec «malloc» (passant par le CPU) et le transfert de données utilisant la mémoire réservée par «cudaHostAlloc()» (sans utilisation du CPU).

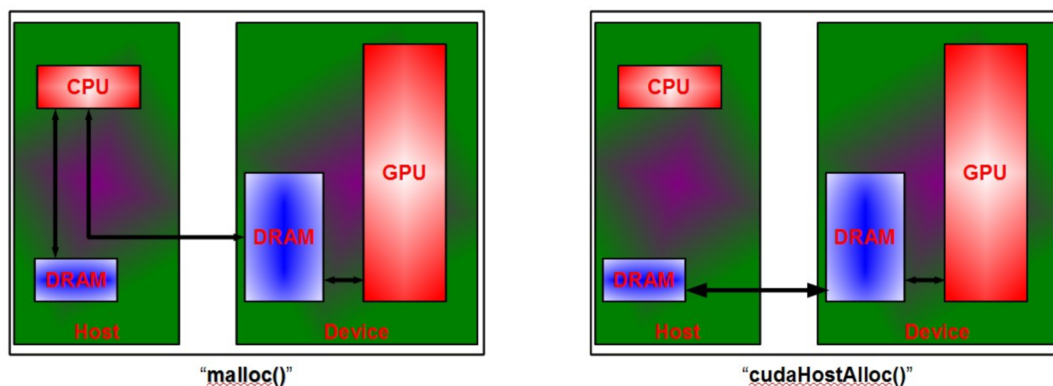


Figure 10 : Type d'accès mémoire

La figure 11 montre que le transfert de données utilisant la DMA peut, pour de vecteurs de taille important, être environ 25% plus rapide.

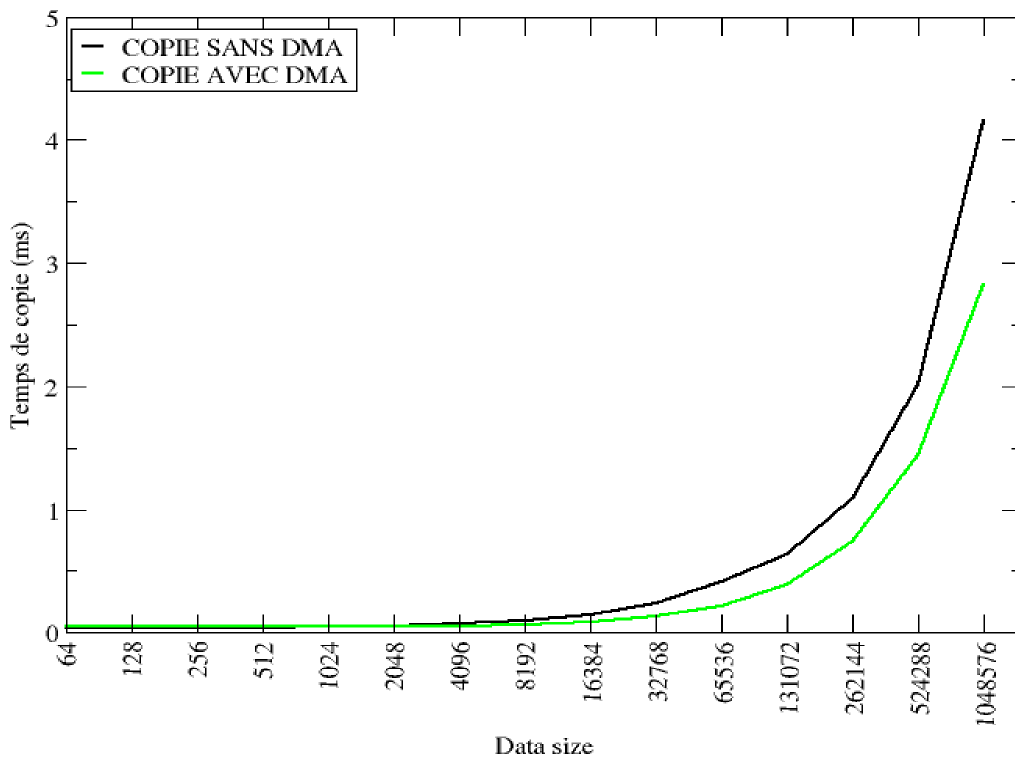


Figure 11 : Le temps de transfert sans/avec DMA vers l'accélérateur GPU

Nous allons nous servir de ce type d'accès mémoire pour la mise en œuvre des transferts de données dans le paragraphe suivant.

### 3.3.2 Transfert synchrone et asynchrone

Il existe 2 modes de transfert de données (voir figure 12) :

- le transfert synchrone : le transfert de données CPU-GPU et GPU-CPU bloque les instructions suivantes sur le CPU. Par contre lors de ce transfert l'exécution sur le GPU (Device) n'est pas bloquant.
- Le transfert asynchrone : ni le transfert de données CPU-GPU ou GPU-CPU, ni l'exécution des calculs (du kernel) sur le GPU (Device) ne sont pas bloquants. Donc le transfert est lancé et sans attendre sa fin, le CPU exécute les instructions suivantes. De plus avec le transfert asynchrone, on peut définir de streams permettant le chevauchement des opérations de transfert et d'exécution sur le GPU.

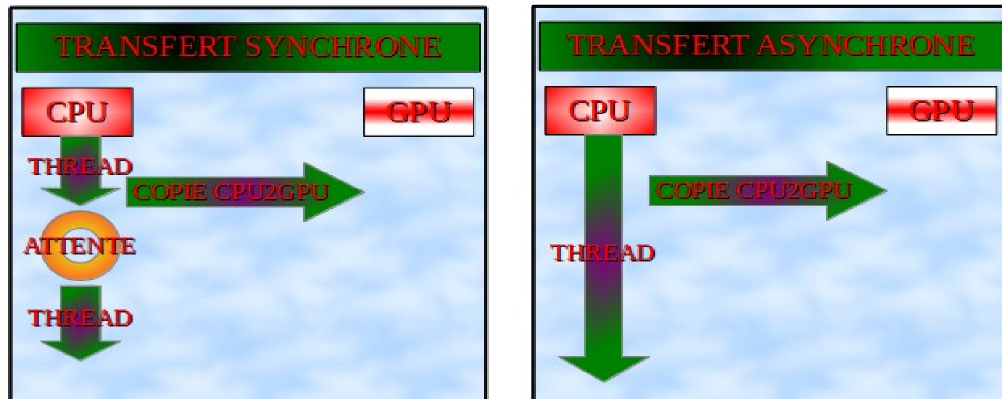


Figure 12 : Fonction des transferts

Comme le GPU (Device) «C2050)» de la Grappe Massivement Parallèle de Calcul Scientifique permet une exécution concurrentielle des opérations, il est possible de chevaucher l'exécution des calculs (du kernel) sur le Device (GPU) avec les transferts mémoires (voir la figure 13 pour l'illustration de l'utilisation de streams).

Pour réaliser le chevauchement de données sur le C2050, la fonction de transfert asynchrone «cudaMemcpyAsync()» utilise un tableau de streams et de la mémoire DMA/pinned. À la fin de l'exécution des opérations asynchrones avec chevauchement de données, une fonction de synchronisation implicite permettant de synchroniser le processeur CPU et l'accélérateur GPU est nécessaire afin de s'assurer que les différentes opérations sur les streams aient fini leurs exécutions. Ce transfert asynchrone avec chevauchement de données permet d'exploiter efficacement la puissance de calcul du Device (GPU) en effectuant du transfert sur une partie de données et du calcul sur le Device (GPU) sur une autre partie de données.

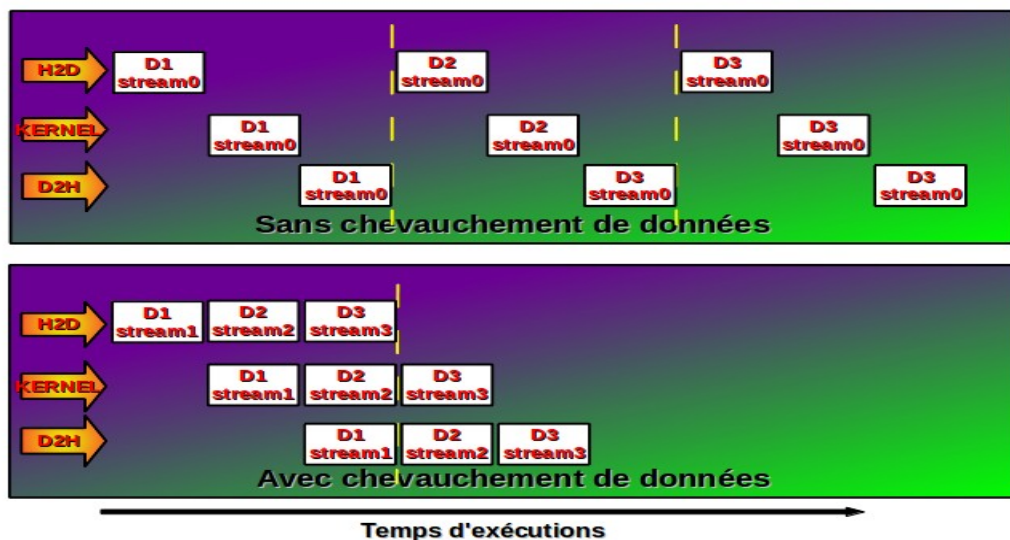


Figure 13 : La ligne de temps d'exécution sur le C2050

Concrètement, dans le cadre de mon stage, la seconde étape de ma réalisation après avoir déterminé le type d'accès mémoire performant concernait tout d'abord :

- dans un premier temps : programmer des opérations synchrones et asynchrone pour la FFT,
- dans un second temps : optimiser le nombre de streams des opérations asynchrones à travers la mise en œuvre du chevauchement de données.

La figure 14 présente l'accélération obtenue lors d'un transfert asynchrone sans chevauchement de données par rapport à un transfert synchrone.

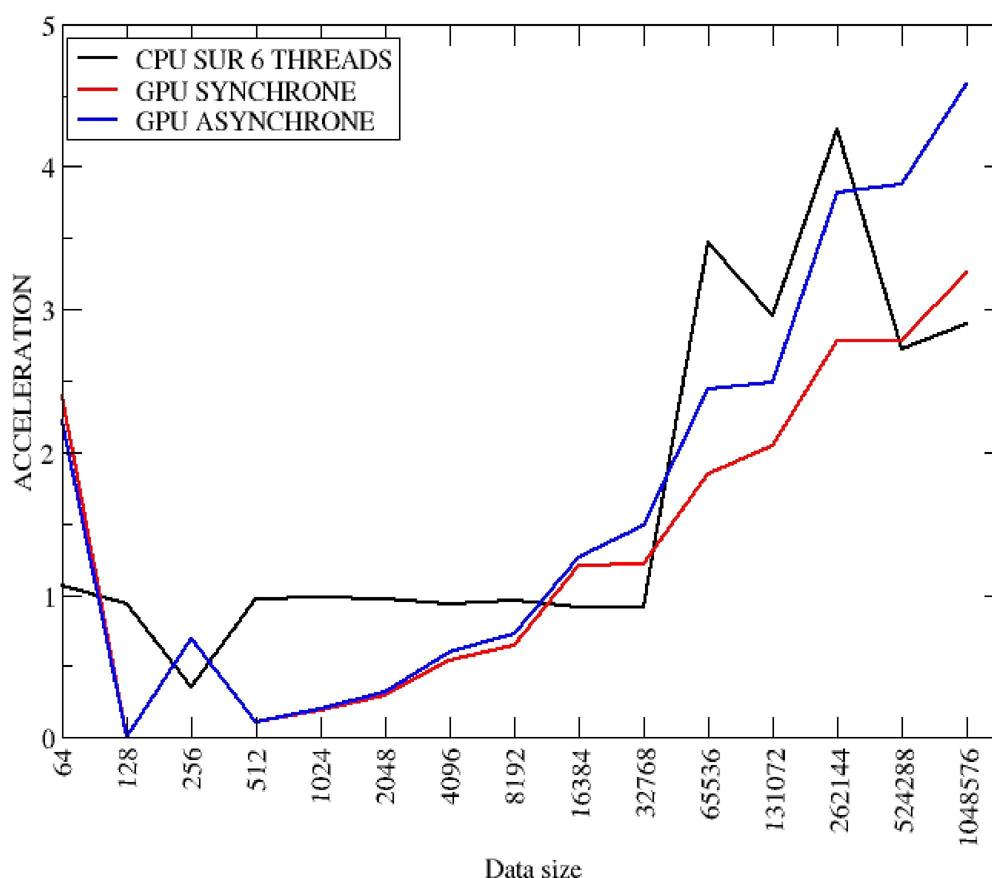


Figure 14: Calcul de la FFT
L'accélération est mesurée par rapport à la FFT exécutée sur 1 cœur du CPU

Nous pouvons remarquer que au fur et à mesure que le nombre de points augmente, l'accélération apportée en utilisant les opérations dites asynchrone est beaucoup plus intéressante que celle des opérations synchrones.

La mise en marche du transfert asynchrone avec chevauchement de données peut être décomposée de la manière suivantes:

- l'allocation de pinned/DMA memory
- la création d'un tableau de streams
- un signal de synchronisation à la fin du transfert asynchrone avec streams

Dans un stream, les opérations sont exécutées d'une manière séquentielle, les unes après les autres dans l'ordre dans lequel elles ont été créées. Alloué à l'aide de la fonction cudaHostAlloc(), le pinned memory garantit au GPU que le système d'exploitation ne paginera pas cet espace mémoire alloué en dehors de la RAM, assurant ainsi le maintient de ce dernier dans la mémoire physique.

A la fin des opérations asynchrones utilisant un tableau de streams, le CPU (Host) lance un signal de synchronisation pour assurer que tous les résultats sont disponibles au niveau de la RAM afin de lancer les opérations suivantes.

La figure 15 présente les résultats des opérations de la FFT effectuées sur le GPU à l'aide de transfert asynchrone avec chevauchement de données utilisant un tableau de streams.

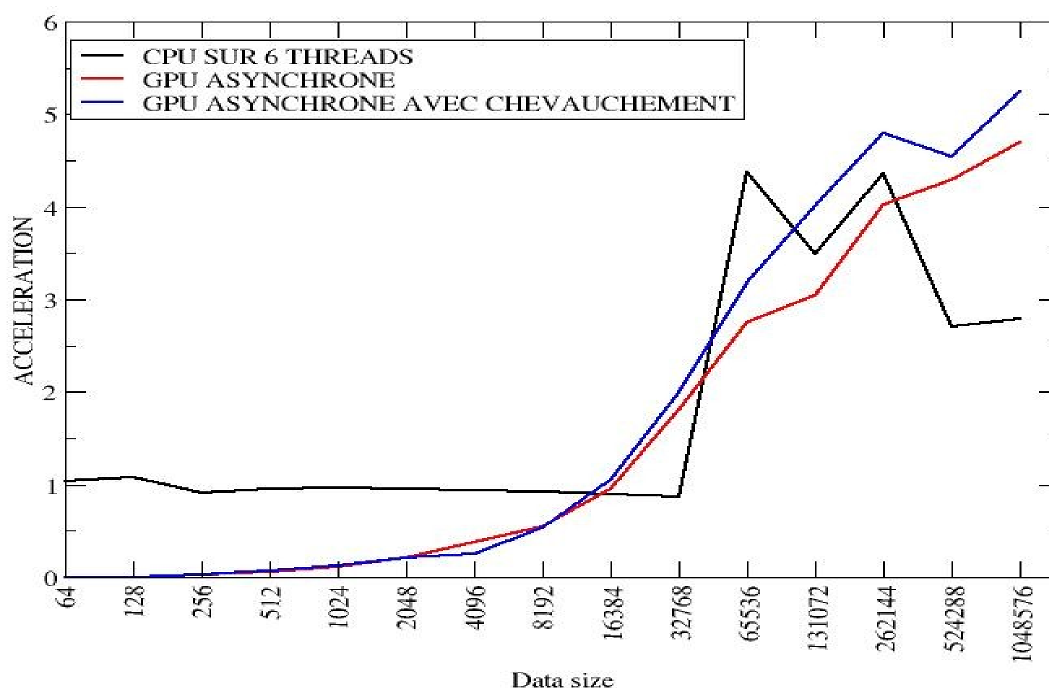


Figure 15 : Optimisations des opérations sur les données
L'accélération est mesurée par rapport à la FFT exécutée sur 1 cœur du CPU

Nous pouvons remarquer que le gain qui est apporté par l'usage du chevauchement de données permet d'exploiter davantage la puissance de calcul du GPU. Cette façon de travailler est vraiment efficace pour les données de grande taille, étant donné que dans ce cas la mémoire cache du CPU (Host) déborde et le calcul sur le Host prend plus de temps pour chercher les données en mémoire RAM du Host.

### 3.4 Cas réel : calcul FFT sur GPU avec transfert de données

En comparant les résultats présentés sur les figures 14 et 15 nous constatons que le gain pour le transfert asynchrone sans et avec chevauchement de données par rapport au transfert synchrone est respectivement de 39 et 55 %. Si l'on compare les transferts asynchrone entre eux le gain est de 16%.

Pour conclure, la figure 16 présente un résumé de l'ensemble de travaux d'optimisation réalisées :

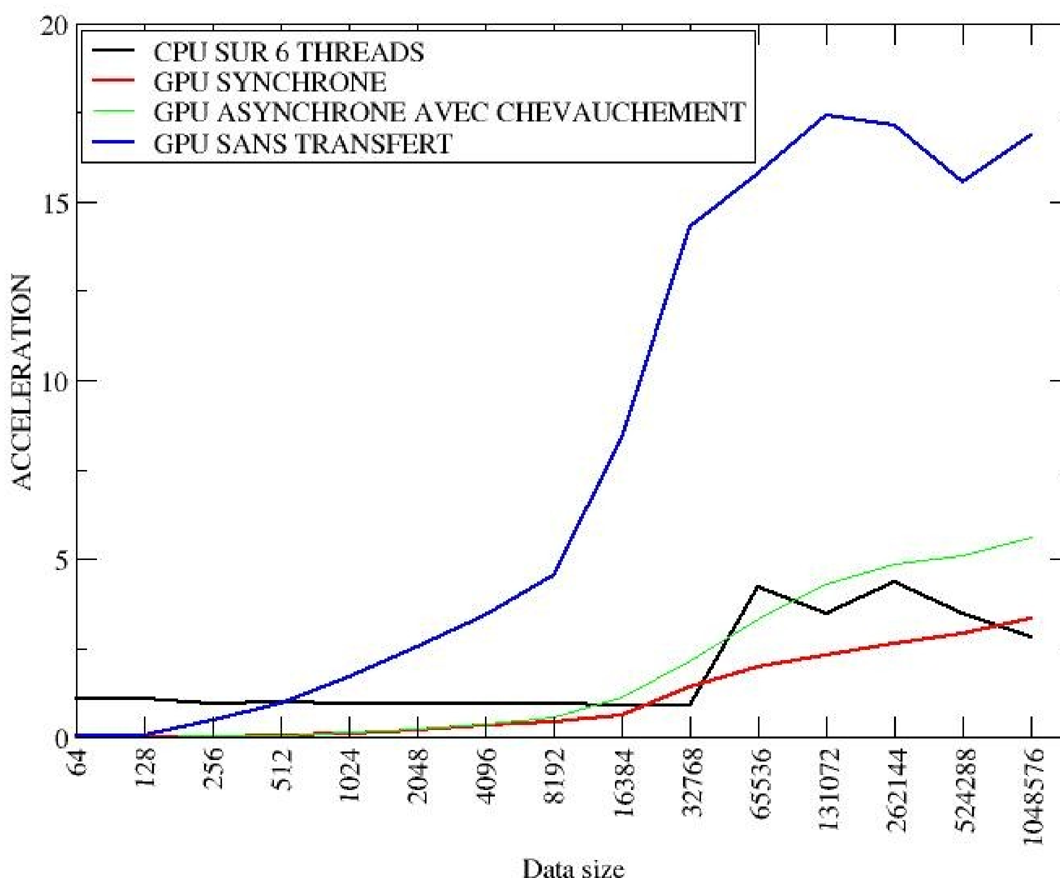


Figure 16 : Bilan sur les optimisations
L'accélération est mesurée par rapport à la FFT exécutée sur 1 cœur du CPU

En effet, nous avons commencé par utiliser l'accélérateur GPU d'une manière synchrone (courbe en rouge sur la figure 15). Ce mode d'exécution s'est révélé inefficace du point de vue exploitation de la puissance de calcul sur l'accélérateur GPU. Avec l'optimisation apportée par le transfert asynchrone avec chevauchement de données, représenté en vert sur la figure, nous sommes arrivés à monter en performance. Par contre, nous sommes encore très loin d'atteindre la puissance de l'accélérateur GPU sans transfert représentée en bleu sur la figure (voir aussi la figure 9) qui est 3 fois supérieur à un calcul réel avec transfert efficace de données CPU-GPU.

## Conclusion

Ce stage s'est intéressé à l'utilisation de l'accélérateur GPU/Nvidia présent sur certains nœuds de calcul de la plate-forme GMPCS de la fédération LUMAT. L'objectif du stage a été d'explorer pour un algorithme particulier (dans notre cas la FFT) les accélérations pouvant être réalisées par rapport à un calcul avec un cœur d'un processeur CPU.

Nous avons constaté que l'étape bloquante dans l'utilisation de l'accélérateur GPU est le transfert de données entre le CPU (Host) et le GPU (Device). Nous avons exploré plusieurs modes de transfert : synchrone, asynchrone et asynchrone avec chevauchement de données. Il s'est avéré que le transfert de données asynchrone avec chevauchement de données est de loin le plus efficace. Mais ce calcul réel reste loin de la puissance théorique du GPU (sans transfert de données) qui est environ 3 fois supérieure.

## Références bibliographiques et numériques


[1] CW,TB,JV,GZ, CUDA C Best Practices Guide v4.1, 01/2012, p21-45, consulté le 31/05/2012 à l'adresse http://developer.nvidia.com/cuda-downloads.

[2] NVIDIA CUDA C Programming Guide v4.2, 16/04/2012, p2-60, consulté le 31/05/2012 à l'adresse http://developer.nvidia.com/cuda-downloads.

[3] CUDA Getting Started Linux v01, 01/2012, p2-9, consulté le 31/05/2012 à l'adresse http://developer.nvidia.com/cuda-downloads.

[4] Bedrich Benes, TECH621GPGPU-05 CUDA Graphics Interoperation, Perdue University, p2-4 consulté le 03/07/2012 à l'adresse http://http://www.purdue.edu

[5] CUDA Toolkit 4.1 CUFFT Library, 01/2012, p24-29, consulté le 25/06/2012 à l'adresse http://developer.nvidia.com/cuda-downloads

[6] Miguel Cadenas Montes, Streams- Learning CUDA to Solve Scientific Problems, Centro de l'investigaciones Medioambientales y Tecnologicas, Madrid, 2010, p2-7, consulté le 03/07/2012 à l'adresse http://http://www.ciemat.es

[7] Florent DAHM, Workshop HPC et GPU Computing, CS Communication & Systèmes, Paris, 06/2012, p3-11, consulté le 10/08/2012 (document reçu par émail après le seminaire au de l'entreprise CS Informatique)

[8] Yukai Hung, Parallel Concept and Hardware Architecture CUDA Programming Model Overview, National Taiwan University, p19-30, consulté le 13/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[9] Yukai Hung , cuda_02_ykhung.pdf, National Taiwan University, p2-5, consulté le 13/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[10] Yukai Hung , cuda_03_ykhung.pdf, National Taiwan University, p10-15, consulté le 13/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[11] Yukai Hung , cuda_04_ykhung.pdf, National Taiwan University, p6-8, consulté le 14/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[12] Yukai Hung , cuda_05_ykhung.pdf, National Taiwan University, p20-22, consulté le 14/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[13] Yukai Hung , cuda_06_ykhung.pdf, National Taiwan University, p3-32, consulté le 15/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[14] Yukai Hung , cuda_07_ykhung.pdf, National Taiwan University, p7-20, consulté le 16/06/2012 à l'adresse http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/

[15] Matthieu Kuhn, La lettre IN2P3 Informatique, Réseau des Informaticiens de l'IN2P3 et de l'IRFU, septembre, numéro 13, 2010, consulté le 23/08/2012 à l'adresse http://informatique.in2p3.fr/li/spip.php?article128

[16] Architecture Intel Many Integrated Core, consulté le 23/08/2012 à l'adresse http://www.intel.fr/content/www/fr/fr/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[17] nvidia_gpu_roadmap.png, consulté le 03/07/2012 à l'adresse http://www.info.univ-angers.fr/pub/richer/cuda_crs4.php

[18] Paul Bourke, Discrete fourier transform – Fast Fourier Transfort, 2003, p1-9, consulté le 20/7/2012 à l'adresse https://local.wasp.uwa.edu.au/~pbourke/miscellaneo

[19] Le GPU (graphic processor unit) ou processeur graphique : carte graphique sur Tom's Guide, consulté le 14/07/2012 à l'adresse http://www.tomsguide.fr/guide/base-Carte-graphique,4-aWRHdWlkZT0zMCZpZENsYXNzZXVyPTUxJmlkUnVicmlxdWU9MjQx.html

[20] Saul A. Teukolsky, Numerical Recipes in Fortran 77 v.1, Cornell University, New York, 1985, p490-525

[21] M. Christian Boulet, Rapport d'évaluation unité de recherche : Fédération de Recherche LUMAT – FR 2764, Université Paris Sud, Orsay, 2009, p5-6

[22] Anthun Fernandes David, Veille Technologique Session Développeur Logiciel 2009, p2-19, consulté le 26/07/2012 à l'adresse : https://docs.google.com/bryss.fr/Afpa/veille/CUDA.ppt

## Annexes

Voici l'ensemble des codes des différentes fonctions utilisées tout au long de mon stage:

Les bibliothèques utilisées

```
/* System standard headers*/
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <time.h>

/* CUFFT headers */
#include <cufft.h>

/* Functions to check execution errors */
#include <cutil_inline.h>
```

1. Cette fonction permet de mesurer le temps de transfert de données synchrone/asynchrone (appel avec Stream = 0)

```
struct event_pair
{
  cudaEvent_t start;
  cudaEvent_t end;
};
inline void start_timer(event_pair * p,cudaStream_t stream[], int nb_elem)
{
  cudaEventCreate(&p->start);
  cudaEventCreate(&p->end);
  cudaEventRecord(p->start,stream[nb_elem]);
};

inline void start_timer(event_pair * p)
{
  cudaEventCreate(&p->start);
  cudaEventCreate(&p->end);
  cudaEventRecord(p->start, 0);
};

inline void stop_timer(event_pair * p,char * kernel_name)
{
  cudaEventRecord(p->end, 0);
  cudaEventSynchronize(p->end);

  float elapsed_time;
  cudaEventElapsedTime(&elapsed_time, p->start, p->end);
  printf("%s prend %0.15f ms\n",kernel_name, elapsed_time/10);
```

```
  cudaEventDestroy(p->start);
  cudaEventDestroy(p->end);
};
```

2. Cette fonction permet de mesurer le temps de transfert de données synchrone/asynchrone (appel avec Stream différent de 0)

```
struct event_pair
{
  cudaEvent_t start;
  cudaEvent_t end;
};
inline void start_timer(event_pair * p,cudaStream_t stream[], int nb_elem)
{
  cudaEventCreate(&p->start);
  cudaEventCreate(&p->end);
  cudaEventRecord(p->start,stream[nb_elem]);
};

inline void stop_timer(event_pair * p, char * kernel_name,cudaStream_t stream[], int nb_elem)
{
      cudaEventRecord(p->end,stream[nb_elem]);
      cudaStreamWaitEvent(stream[nb_elem],p->end,0);

  float elapsed_time;
  cudaEventElapsedTime(&elapsed_time, p->start, p->end);
  printf("%s prend %0.9f ms\n",kernel_name, elapsed_time/10);
  cudaEventDestroy(p->start);
  cudaEventDestroy(p->end);
};
```

3. La fonction Synchrone (Sync)

```
cufftResult zfft1d_gpu_Sync(int nx,int nb_Stream,int nb_ifft_fft,cufftDoubleComplex
*host_in,cufftDoubleComplex *host_out)
{
   //* number of transforms of size nx
   int BATCH = 1;
   int loop;
   int offset;
   int snum = nb_Stream;
   int taille_plan = nx/snum;
   int cufft_forward_or_backforward = nb_ifft_fft;

  //* creation des variables pour mesurer le temps d'exécution
   event_pair start_stop_DStream;

   //* declaration of GPU arrays
```

```
  cufftDoubleComplex *device_mem_in;
  cufftDoubleComplex *device_mem_out;

  //* compute the size of the arrays in bytes
  int num_bytes =  nx * sizeof(cufftDoubleComplex);
  int stream_num_bytes = num_bytes/snum;

  //* cudaMalloc device arrays
  //* if memory allocation failed, report an error message and exit
  cutilSafeCall(cudaMalloc((void**)&device_mem_in, num_bytes));
  cutilSafeCall(cudaMalloc((void**)&device_mem_out, num_bytes));


//*              Debut du procedure de la transformée Fourier
//*                  ainsi que de la transformée inverse

int tmp = 0;
  start_timer(&start_stop_DStream);
while(tmp<10)
 {

  //* copy data to GPU
  //* if CPU to GPU copy failed, report an error message and exit
  for(loop=0;loop<snum;loop++)
  {
    offset = (loop)*nx/snum;
            cutilSafeCall(cudaMemcpy(device_mem_in+offset,     host_in+offset,stream_num_bytes,
cudaMemcpyHostToDevice));
  }

  //* create a 1D FFT plan: plan adapté à la dimension de chaque partie de vecteur, d'où le size dans
fonction
  cufftHandle  plan;
  cufftSafeCall(cufftPlan1d(&plan, taille_plan ,CUFFT_Z2Z , BATCH));
    //cufftSetStream(plan,0);  // très important pour l'exécution de stream les un après les autres
                             // ou pour le chevauchement des données
  for(loop=0;loop<snum;loop++)
  {
    offset = (loop)*nx/snum;
  if(cufft_forward_or_backforward ==1)
  //* compute FFT FORWARD
          cufftSafeCall(cufftExecZ2Z    (plan,    device_mem_in+offset,    device_mem_out+offset,
CUFFT_FORWARD));
  else
  //* compute FFT INVERSE
          cufftSafeCall(cufftExecZ2Z    (plan,    device_mem_in+offset,    device_mem_out+offset,
CUFFT_INVERSE));
  }
```

```
  for(loop=0;loop<snum;loop++)
  {
    offset = (loop)*nx/snum;
  //* copy data back to CPU
  //* if GPU to CPU copy failed, report an error message and exit
        cutilSafeCall(cudaMemcpy(host_out+offset, device_mem_out+offset, stream_num_bytes,
cudaMemcpyDeviceToHost));
  }
  //* destroy the CUFFT plan
   cufftSafeCall(cufftDestroy(plan));
 tmp++;
 }
 stop_timer(&start_stop_DStream, "le temps d'exécution sur le GPU Synchrone");

  //* deallocate memory
  cutilSafeCall(cudaFree(device_mem_in));
  cutilSafeCall(cudaFree(device_mem_out));

  //* return value in case of success
  return CUFFT_SUCCESS;
};
```

3. La fonction Asynchrone (Async_S_Ch)

```
cufftResult zfft1d_gpu_Async_S_Ch(int nx,int nb_Stream,int nb_ifft_fft,cufftDoubleComplex
*host_in,cufftDoubleComplex *host_out)
{
  //* number of transforms of size nx
  int BATCH = 1;
  int loop;
  int offset;
  int snum = nb_Stream;
  int taille_plan = nx/snum;
  int cufft_forward_or_backforward = nb_ifft_fft;


 //* creation des variables pour mesurer le temps d'exécution
  event_pair start_stop_DStream;

  //* declaration of GPU arrays
  cufftDoubleComplex *device_mem_in;
  cufftDoubleComplex *device_mem_out;

  //* compute the size of the arrays in bytes
  int num_bytes =  nx * sizeof(cufftDoubleComplex);
  int stream_num_bytes = num_bytes/snum;
```

```
  //* cudaMalloc device arrays
  //* if memory allocation failed, report an error message and exit
  cutilSafeCall(cudaMalloc((void**)&device_mem_in, num_bytes));
  cutilSafeCall(cudaMalloc((void**)&device_mem_out, num_bytes));



//*            Debut du procedure de la transformée Fourier
//*                ainsi que de la transformée inverse

int tmp = 0;
  start_timer(&start_stop_DStream);
while(tmp<10)
 {
  for(loop=0;loop<snum;loop++)
  {
    offset = (loop)*nx/snum;


  //* copy data to GPU
  //* if CPU to GPU copy failed, report an error message and exit

      cutilSafeCall(cudaMemcpyAsync(device_mem_in+offset,   host_in+offset,stream_num_bytes,
cudaMemcpyHostToDevice,0));


  //* create a 1D FFT plan: plan adapté à la dimension de chaque partie de vecteur, d'où le size dans
fonction
  cufftHandle  plan;
  cufftSafeCall(cufftPlan1d(&plan, taille_plan ,CUFFT_Z2Z , BATCH));
  cufftSetStream(plan,0);  // très important pour l'exécution de stream les un après les autres
                           // ou pour le chevauchement des données
  if(cufft_forward_or_backforward ==1)
  //* compute FFT FORWARD
          cufftSafeCall(cufftExecZ2Z    (plan,    device_mem_in+offset,    device_mem_out+offset,
CUFFT_FORWARD));
  else
  //* compute FFT INVERSE
          cufftSafeCall(cufftExecZ2Z    (plan,    device_mem_in+offset,    device_mem_out+offset,
CUFFT_INVERSE));

  //* copy data back to CPU
  //* if GPU to CPU copy failed, report an error message and exit
   cutilSafeCall(cudaMemcpyAsync(host_out+offset,  device_mem_out+offset,  stream_num_bytes,
cudaMemcpyDeviceToHost,0));
  //* destroy the CUFFT plan
  cufftSafeCall(cufftDestroy(plan));
  }
```

```
 tmp++;
 }
  stop_timer(&start_stop_DStream, "le temps d'exécution sur le GPU Asynchrone(sans chevauchement des données)");

  //* deallocate memory
  cutilSafeCall(cudaFree(device_mem_in));
  cutilSafeCall(cudaFree(device_mem_out));


  //* return value in case of success
  return CUFFT_SUCCESS;
};
```

4. La fonction Asynchrone Avec chevauchement (Async_A_Ch)

```
cufftResult zfft1d_gpu_Async_A_Ch(int nx,int nb_Stream,int nb_ifft_fft,cufftDoubleComplex *host_in,cufftDoubleComplex *host_out)
{
  //* number of transforms of size nx
  int BATCH = 1;
  int loop;
  //int offset;
  int snum = nb_Stream;
  int taille_plan = nx/snum;
  int cufft_forward_or_backforward = nb_ifft_fft;

 //* creation des variables pour mesurer le temps d'exécution
  event_pair start_stop_Stream;//t2,t1,t3;

  //* declaration of GPU arrays
  cufftDoubleComplex *device_mem_in;
  cufftDoubleComplex *device_mem_out;

  //* compute the size of the arrays in bytes
  int num_bytes =  nx * sizeof(cufftDoubleComplex);
  int stream_num_bytes = num_bytes/snum;

  //* cudaMalloc device arrays
  //* if memory allocation failed, report an error message and exit
  cutilSafeCall(cudaMalloc((void**)&device_mem_in, num_bytes));
  cutilSafeCall(cudaMalloc((void**)&device_mem_out, num_bytes));


 //*Creation of a stream Id
 cudaStream_t stream[snum];
 cudaEvent_t cycleDone[snum];
```

```
  for(loop=0;loop<snum;loop++)
    {
      cudaStreamCreate(&stream[loop]);
      cudaEventCreate(&cycleDone[loop]);
    }
/*
    cudaEvent_t start_event, stop_event;
    float time_memcpy;
    int  device_sync_method = cudaDeviceBlockingSync;
         int    eventflags    =    (    (device_sync_method    ==    cudaDeviceBlockingSync)    ?
cudaEventBlockingSync: cudaEventDefault );
    cudaEventCreateWithFlags(&start_event, eventflags);
    cudaEventCreateWithFlags(&stop_event, eventflags);
    cudaEventRecord(start_event, 0);
*/

  int tmp = 0;
   start_timer(&start_stop_Stream);
 while(tmp<10)
 {
   int present_offset = 0;
                              cutilSafeCall(cudaMemcpyAsync(device_mem_in+present_offset,
host_in+present_offset,stream_num_bytes, cudaMemcpyHostToDevice,stream[0]));
  for(loop=0;loop<snum;loop++)
  {
     int j=1;
     int prochain_offset = (j)*nx/snum;

   //* copy data to GPU
   //* if CPU to GPU copy failed, report an error message and exit
   //start_timer(&t1);
      //cutilSafeCall(cudaMemcpyAsync(device_mem_in+offset,  host_in+offset,stream_num_bytes,
cudaMemcpyHostToDevice,stream[loop+1]));

   //stop_timer(&t1, "le temps de copie CPU_GPU(avec chevauchement des données)");
   //* create a 1D FFT plan
   cufftHandle  plan;
   cufftSafeCall(cufftPlan1d(&plan, taille_plan ,CUFFT_Z2Z , BATCH));
   cufftSetStream(plan,stream[loop]);// pour l'exécution de stream les un après les autres
                        // ou pour le chevauchement des données
   //start_timer(&t2);
   if(cufft_forward_or_backforward ==1)
   //* compute FFT FORWARD
          cufftSafeCall(cufftExecZ2Z    (plan,    device_mem_in+present_offset,    device_mem_out,
CUFFT_FORWARD));
   else
   //* compute FFT INVERSE
                    cufftSafeCall(cufftExecZ2Z        (plan,        device_mem_in+present_offset,
```

```
device_mem_out+present_offset, CUFFT_INVERSE));
  //stop_timer(&t2, "le temps de Kernel(avec chevauchement des données)");
  //* copy data back to CPU
  //* if GPU to CPU copy failed, report an error message and exit
  //start_timer(&t3);
  if(loop<(snum-1))
                                        cutilSafeCall(cudaMemcpyAsync(device_mem_in+prochain_offset,
host_in+prochain_offset,stream_num_bytes, cudaMemcpyHostToDevice,stream[loop+1]));

      cutilSafeCall(cudaMemcpyAsync(host_out+present_offset, device_mem_out+present_offset,
stream_num_bytes, cudaMemcpyDeviceToHost,stream[loop]));

  cudaEventRecord(cycleDone[loop],stream[loop]);
  present_offset = prochain_offset;

  //stop_timer(&t3, "le temps de copie GPU_CPU(avec chevauchement des données)");
  //* destroy the CUFFT plan
  j++;
  cufftSafeCall(cufftDestroy(plan));
  }
 tmp++;
 }
   stop_timer(&start_stop_Stream, "le temps d'exécution sur le GPU Asynchrone(avec
chevauchement des données)");

/*
   cudaEventRecord(stop_event, 0);
   cudaEventSynchronize(stop_event);   // block until the event is actually recorded
   cudaEventElapsedTime(&time_memcpy, start_event, stop_event);
   printf("temps d'exécution:\t%.2f\n", time_memcpy);
*/

  //* deallocate memory
  cutilSafeCall(cudaFree(device_mem_in));
  cutilSafeCall(cudaFree(device_mem_out));

  //* stream destruction
  for(loop=0;loop<snum;loop++)
   {
    cudaStreamDestroy(stream[loop]);
    cudaEventDestroy(cycleDone[loop]);
   }

  //* return value in case of success
  return CUFFT_SUCCESS;
};
```

5. La fonction d'appel de la FFT utilisant la mémoire vive

```
void IFFT_FFT_APPEL_MALLOC(int data_Length, int nb_Stream,int nb_ifft_fft)
{
//Attention: si la valeur de data_Length est un pair, nb_Stream doit imperativement l'être aussi
//         et la valeur de data_Length est un impair, nb_Stream doit imperativement l'être aussi
   int vSize,snum;
   vSize = data_Length*nb_Stream;
   snum = nb_Stream;
   int i,j;
   int nBytes;
   int off_set = vSize/snum;
   int m_nb_ifft_fft;
   cufftDoubleComplex *data_in,*data_out;
   cufftResult returnValue;
   m_nb_ifft_fft = nb_ifft_fft;
//  -si jamais la valeur de data_Length est pair et que celle de nb_Stream impair,
// on prendre alors comme nb_Stream = 1,ceci pour éviter une division inéquitable
// de notre vecteur en sous vecteur.
//  -si jamais l'utilisateur m_nb_ifft_fft=0, alors m_nb_ifft_fft=1,car la ifft(fft)
// doit au moins s'exécuter une fois
   if((vSize%snum)!=0)
   {
    snum = 1;
   }

   if(m_nb_ifft_fft==0)
   {
     m_nb_ifft_fft = 1;
   }
// Host allocation memory
   nBytes = vSize*sizeof(cufftDoubleComplex);
   data_in = (cufftDoubleComplex*)malloc(nBytes);
   data_out = (cufftDoubleComplex*)malloc(nBytes);

// decoupage du tableau en 3 parties etpuis remplissage de ce dernier
  j=0;
  int k=1;

 while(j<vSize)
  {
    for(i=0;i<off_set;i++)
     {
       data_in[i+j].x = k;
       data_in[i+j].y = 0;
     }
     k++;
     j+=(off_set);
  }
```

```
// Appel de la fonction IFFT(FFT())
  returnValue = zfft1d_gpu_Sync(vSize,snum,m_nb_ifft_fft,data_in,data_out);

  if (returnValue != CUFFT_SUCCESS)
    fprintf (stderr, "Error performing function zfft1d_gpu\n");
  else
    fprintf (stdout, "All is well !\n");

// Vérification des resultas
 /*for(i=0;i<vSize;i++)
    {
     if((data_out[i].x != data_in[i].x * off_set)&&(data_out[i].y !=0))
     exit(1);
    }

// Affichage des résultats
 for (i=0; i<vSize; i++) {
    printf ("Data_out: Real part of element %d is %f\n", i, data_out[i].x);
    printf ("        Imag part of element %d is %f\n\n", i, data_out[i].y);
  }*/
   free(data_in);
   free(data_out);
};
```

6. La fonction d'appel de la FFT utilisant le page locked memory (pinned memory)

```
void IFFT_FFT_APPEL_PINNED(int data_Length, int nb_Stream,int nb_ifft_fft,int option)
{
//Attention: si la valeur de data_Length est un pair, nb_Stream doit imperativement l'être aussi
//         et la valeur de data_Length est un impair, nb_Stream doit imperativement l'être aussi
  int vSize,snum;
  vSize = data_Length*nb_Stream;
  snum = nb_Stream;
  int i,j;
  int m_option = option;
  int nBytes;
  int off_set = vSize/snum;
  int m_nb_ifft_fft;
  cufftDoubleComplex *data_in,*data_out;
  cufftResult returnValue;
  m_nb_ifft_fft = nb_ifft_fft;
//  -si jamais la valeur de data_Length est pair et que celle de nb_Stream impair,
// on prendre alors comme nb_Stream = 1,ceci pour éviter une division inéquitable
// de notre vecteur en sous vecteur.
//  -si jamais l'utilisateur m_nb_ifft_fft=0, alors m_nb_ifft_fft=1,car la ifft(fft)
// doit au moins s'exécuter une fois
  if((vSize%snum)!=0)
```

```
   {
    snum = 1;
   }

   if(m_nb_ifft_fft==0)
   {
     m_nb_ifft_fft = 1;
   }
// Host allocation memory
   nBytes = vSize*sizeof(cufftDoubleComplex);
   cutilSafeCall(cudaMallocHost((void**)&data_in,nBytes,cudaHostAllocWriteCombined));
   cutilSafeCall(cudaMallocHost((void**)&data_out,nBytes,cudaHostAllocWriteCombined));
// decoupage du tableau en 3 parties etpuis remplissage de ce dernier
  j=0;
  int k=1;

 while(j<vSize)
  {
    for(i=0;i<off_set;i++)
     {
       data_in[i+j].x = k;
       data_in[i+j].y = 0;
     }
     k++;
     j+=(off_set);
  }

// Appel de la fonction IFFT(FFT())
   switch(m_option)
   {
     case 1:
          returnValue = zfft1d_gpu_Async_S_Ch(vSize,snum,m_nb_ifft_fft, data_in,data_out);
          break;
     case 2:
          returnValue = zfft1d_gpu_Async_A_Ch(vSize,snum,m_nb_ifft_fft, data_in,data_out);
          break;
     case 3:
          returnValue = zfft1d_gpu_Sync(vSize,snum,m_nb_ifft_fft, data_in,data_out);
          break;
    default:
          returnValue = zfft1d_gpu_Async_A_Ch(vSize,snum,m_nb_ifft_fft, data_in,data_out);
          break;
   }

   if (returnValue != CUFFT_SUCCESS)
     fprintf (stderr, "Error performing function zfft1d_gpu\n");
   else
     fprintf (stdout, "All is well !\n");
```

```
/*
// Vérification des resultas
 for(i=0;i<vSize;i++)
    {
      if((data_out[i].x != data_in[i].x * off_set)&&(data_out[i].y !=0))
      exit(1);
    }*/

// Affichage des résultats
   /*for (i=0; i<vSize; i++) {
      printf ("Data_out: Real part of element %d is %f\n", i, data_out[i].x);
      printf ("        Imag part of element %d is %f\n\n", i, data_out[i].y);
   }*/
     cudaFreeHost(data_in);
     cudaFreeHost(data_out);

};
```

7. le main permettant d'exécuter les différentes fonctions citées ci-haut

```
int main(int argc, char** argv)
{
     // Attention lors de l'appel de la fonction IFFT_FFT_APPEL_PINNED(int data_Length, int nb_Stream,int nb_ifft_fft,int option)
     // data_Length: c'est la taille du vecteur
    // nb_Stream: c'est le nombre de sous vecteur que comportera le vecteur principal(c.à.d: data_Length/nb_Stream)
    // nb_ifft_fft: sert à choisir, soit la fft où la ifft
    // si jamais nb_ifft_fft=1 alors la fft va s'exécuter, sinon la ifft s'exécutera.
   // option: c'est un paramètre qui permet de selectionner la fonction qu'on veut utiliser
   // si option = 1 : appel de la fonction zfft1d_gpu_Async_S_Ch()
   // si option = 2 : appel de la fonction zfft1d_gpu_Async_A_Ch
   // si option = 3 : appel de la fonction zfft1d_gpu_Sync_Pinned
   // par defaut(si option != 1,2,3): appel de la fonction zfft1d_gpu_Async_A_Ch

  for(int i=6;i<=20;i++)
  {
    int m_taille = 1<<i;
      IFFT_FFT_APPEL_PINNED(m_taille,1,1,3);
      IFFT_FFT_APPEL_PINNED(m_taille,5,1,1);
      IFFT_FFT_APPEL_PINNED(m_taille,10,1,1);
      IFFT_FFT_APPEL_MALLOC(m_taille,1,1);
      IFFT_FFT_APPEL_MALLOC(m_taille,5,1);
      IFFT_FFT_APPEL_MALLOC(m_taille,10,1);
  }

}
```

8. le makefile permettant la compilation

```
# A simple CUDA makefile.
# USAGE:
#  compile:
#       make all        // compiles all the parts
#
#  run:
#       make run1       // runs part 1
#       make run2       // runs part 2
#       make run3       // runs part 3

# CUDA depends on two things:
#  1) The CUDA nvcc compiler, which needs to be on your path,
#       or called directly, which we do here
#  2) The CUDA shared library being available at runtime,
#       which we make available by setting the LD_LIBRARY_PATH
#       variable for the durection of the makefile.
#
# You can set your PATH and LD_LIBRARY_PATH variables as part of your
# .profile so that you can compile and run without using this makefile.

#NVCCFLAGS      :=
#NVCC           := /source /opt/software/nvidia/cuda_env.sh
#LD_LIBRARY_PATH := /usr/local/cuda/lib64


sync1: TestTransfertDonnees.cu gpu_util.h fonction_Test.h
        nvcc TestTransfertDonnees.cu -o TestTransfertDonnees
sync: Sync_Gpu_Cpu.cu gpu_util.h
        nvcc Sync_Gpu_Cpu.cu -o Sync_Gpu_Cpu

async: Async_Gpu_Cpu.cu gpu_util.h
        nvcc Async_Gpu_Cpu.cu -o Async_Gpu_Cpu

asyncStream: Async_UseOfStream_Gpu_Cpu.cu gpu_util.h
        nvcc Async_UseOfStream_Gpu_Cpu.cu -o Async_UseOfStream_Gpu_Cpu

asyncStream1: GPU_FFT_IFFT_1D.cu gpu_util.h
        nvcc -DDEBUG GPU_FFT_IFFT_1D.cu -I/opt/software/nvidia/NVIDIA_GPU_Computing_SDK_4.1.28/C/common/inc/ -lcufft -o GPU_FFT_IFFT_1D

cleanSync1:
        rm -rf TestTransfertDonnees

cleanSync:
        rm -rf Sync_Gpu_Cpu
```

```
cleanAsync:
	rm -rf Async_Gpu_Cpu
cleanStream:
	rm -rf Async_UseOfStream_Gpu_Cpu
cleanStream1:
	rm -rf GPU_FFT_IFFT_1D

runSync1: TestTransfertDonnees
	./TestTransfertDonnees
runSync: Sync_Gpu_Cpu
	./Sync_Gpu_Cpu
runAsync: Async_Gpu_Cpu
	./Async_Gpu_Cpu
runStream: Async_UseOfStream_Gpu_Cpu
	./Async_UseOfStream_Gpu_Cpu
runStream1: GPU_FFT_IFFT_1D
	./GPU_FFT_IFFT_1D
```

9. le fichier de soumission vers le GPU (qpu.qsub)

```
#!/bin/sh

#---------------------------------------------------------------
#	Exemple de directives SGE minimales d'execution d'un job
#---------------------------------------------------------------

#$ -q cuda				#	Pour utiliser cudawrapper dans la queue cuda
					#	(epilog et prolog)

#$ -m base				#	pour quelles actions envoyer un corriel
#$ -M jean.yaokeli-likoke@u-psud.fr	#	courriel où envoyer les informations

#$ -N GPU_job				#	nom du job qui apparaîtra dans la queue
#$ -o GPU_job.out			#	nom du fichier de sortie

#$ -l h_rt=00:30:00			#	on demande 30 minutes de temps de calcul
#$ -l h_vmem=2G				#	on demande 2G de mémoire vivre
#$ -l gpu=1

#$ -cwd					#	passer au répertoire courant
#$ -j y					#	la sortie erreur est fusionnée avec la sortie standard

#---------------------------------------------------------------
#	Fin de l'exemple de directives SGE
#---------------------------------------------------------------
```

```
#-------------- Imprimer le noeud de calcul  ----------------------
hostname -f
#----------------------------------------------------------------

#------------- Imprimer la date et le temps  ----------------------
date
echo "JOB START"
#----------------------------------------------------------------


#----------------------------------------------------------------
#        Début des commandes unix/linux utilisateur permettant de lancer le calcul
#----------------------------------------------------------------

source /opt/software/nvidia/cuda_env.sh

nvcc -V

#nvcc gpu_cpu.cu -o gpu_cpu

#gpu_cpu

#make cleanSync1
#make sync1
#make runSync1

make cleanStream1
make asyncStream1
make runStream1

#export OMP_NUM_THREADS=1
#echo OMP_NUM_THREADS $OMP_NUM_THREADS

#make -f fft_mkl_1D_x.Makefile clean
#make -f fft_mkl_1D_x.Makefile
#make -f fft_mkl_1D_x.Makefile run_mkl_1D_x

#make cleanSync
#make cleanAsync
#make cleanStream
#make sync
#make async
#make asyncStream
#make runSync
#make runAsync
#make runStream
#----------------------------------------------------------------
#make cleanSync
```

```
#make cleanAsync
#make cleanStream
#make sync
#make async
#make asyncStream
#make runSync
#make runAsync
#make runStream
#---------------------------------------------------------------
#       Fin de commandes unix/linux utilisateur
#---------------------------------------------------------------


#------------- Imprimer la date et le temps  ----------------------
echo "JOB END"
date
#---------------------------------------------------------------

##end of sge qsub script file
```