# Étude et tests de CUDA™ (NVIDIA) appliqués à la mise en œuvre d'algorithmes de télécommunications

Jean-Charles Paoletti

le cnam

**CONSERVATOIRE NATIONAL DES ARTS ET METIERS**

**PARIS**

---

MEMOIRE

présenté en vue d'obtenir le

**DIPLOME D'INGENIEUR CNAM**

en

ELECTRONIQUE

PAR

Jean-Charles PAOLETTI

---

# Etude et tests de CUDA™ (NVIDIA) appliqués à la mise en œuvre d'algorithmes de télécommunications

**Soutenu le 05 novembre 2013**

---

**JURY**

| | | |
|---|---|---|
| **PRESIDENT :** | **M. D. LE RUYET** | **Professeur des Universités - CNAM** |
| **MEMBRES :** | **Mme C. ALGANI** | **Professeur des Universités - CNAM** |
| | **M. C. PAUTOT** | **Maître de conférences - CNAM** |
| | **M. S. LETOURNEUR** | **Développeur FPGA - ELSYS DESIGN** |
| | **M. T. DO** | **Ingénieur Electronicien - SAGEM** |

**Remerciements :**

Je remercie le professeur Han VU THIEN de m'avoir accueilli dans son laboratoire pour effectuer mon stage de fin d'études.

Je remercie Christian PAUTOT pour m'avoir proposé ce sujet de mémoire et pour ses conseils pour l'élaboration de ce mémoire. Je remercie également Christophe ALEXANDRE pour ses conseils techniques.

Je remercie l'ensemble des stagiaires du Laboratoire Signaux et Systèmes du CNAM pour leurs nombreux conseils utiles lors de ma recherche d'emploi et pour les bons moments passés ensemble.

Je remercie également ma famille – et en particulier ma sœur Sophie PAOLETTI – pour son soutien moral tout au long de mes études au CNAM.

# Table des matières

# Introduction

Depuis une trentaine d'année, portée par l'industrie des jeux vidéos, la restitution à l'écran de données graphiques est une des fonctions informatiques qui a le plus évolué.

Pendant longtemps, la restitution graphique de données vidéo et la création de graphismes pour les jeux vidéo ont été mises en œuvre dans les cartes graphiques par ce qu'on appelle un pipeline graphique. Un pipeline graphique est un ensemble d'étapes reliées ensemble de manière fixe. Les résultats d'une étape étaient utilisés comme données d'entrée pour l'étape suivante et à chaque étape était dédié un programme et/ou des ressources fixes.

Puis, poussée par une demande en progression constante, l'industrie des cartes graphiques s'est énormément développée afin de satisfaire principalement les joueurs de jeu vidéo en recherche permanente de graphismes proches du réel. Les pipelines des cartes graphiques se sont donc progressivement complexifiés avec de plus en plus d'étapes et sous-étapes mises en œuvre dans les versions de l'API graphique DirectX. Les cartes graphiques ont rapidement atteint des puissances de calcul telles que les scientifiques ont commencé à s'y intéresser non plus pour faire du traitement graphique mais pour exécuter des calculs séparables en plusieurs tâches identiques attribuées à chacun des cœurs du processeur graphique (GPU). C'est ce qu'on a appelé le GPGPU [1] – abbréviation de General-Purpose processing on Graphics Processing Units – c'est à dire du calcul générique sur processeur graphique.

Puis fin 2006, environ un an avant son principal concurrent (la société canadienne ATI devenue AMD), la société NVIDIA change complètement – avec son processeur G80 – la manière dont sont mis en œuvre les pipelines dans les cartes graphiques. Désormais, les ressources de calcul fixes sont remplacées par des ressources de calcul réutilisables par chaque étape d'un traitement graphique. C'est ce qu'on appellera l'architecture dite « unifiée ». Désormais, le calcul scientifique est pris en compte à part entière à travers une architecture de calcul (appelée « pipeline de calcul » sur le G80 [2]) dédiée, basée sur les mêmes cœurs que l'architecture graphique.

En 2008, NVIDIA introduit une version très améliorée du processeur G80, c'est le processeur GT200. L'architecture des processeurs G80 et GT200 était appelée architecture « Tesla » [3][4]. Comme pour le processeur G80, l'architecture de traitement graphique et l'architecture de calcul du processeur GT200 utilisent les mêmes cœurs pour fonctionner.

Puis une architecture totalement nouvelle basée sur les demandes des développeurs fait son apparition en 2010, c'est l'architecture Fermi mise en œuvre dans le processeur GF100. Avec le modèle Fermi, se côtoient une architecture graphique basée sur l'API DirectX 11 et une architecture de calcul.

Le processeur graphique présent sur chaque carte graphique est divisé en un certain nombre de processeurs de calcul appelés cœurs, contenant les ressources minimales nécessaires au traitement graphique. Ces cœurs verront leur nombre se multiplier en très peu d'années. Les processeurs NVIDIA les plus puissants comportent actuellement (2011) 512 cœurs ce qui les rend plus rapides que le CPU lui même pour un certain nombre de calculs. L'objet de ce mémoire sera donc d'étudier les capacités de calcul d'une des cartes graphiques NVIDIA dernière génération, la GTX 470, comprenant

448 cœurs et basée sur l'architecture Fermi. Le thème des cartes graphiques étant très vaste, nous nous limiterons à l'étude des cartes NVIDIA, bien que les cartes AMD (anciennement ATI) possèdent des capacités de calcul tout à fait comparables aux cartes NVIDIA.

Dès les débuts de l'architecture unifiée NVIDIA fin 2006, un environnement de programmation est dédié aux calculs sur carte graphique, c'est l'environnement CUDA. A chaque nouvelle architecture (GT200 et GF100), des bibliothèques de fonctions nouvelles apparaissent.

Dans ce mémoire, nous étudierons dans un premier temps :

- un exemple de programmation CUDA,
- puis l'intégration d'un programme CUDA dans une fonction MEX (pour que la fonction CUDA-C puisse communiquer avec MATLAB),
- et enfin la mise en œuvre de CUDA dans un calcul de multiplication de matrices.

Nous étudierons également la bibliothèque CUBLAS dans un programme :

- simple,
- puis intégré à une fonction MEX,
- et enfin mise en œuvre dans un calcul de multiplication de matrice.

Nous comparerons les performances :

- de chaque API selon les variations de certains paramètres,
- puis des API CUDA et CUBLAS,
- et enfin des deux API par rapport à MATLAB utilisé seul.

Puis nous étudierons l'API CUFFT :

- dans un exemple simple,
- puis dans un programme plus complexe intégré à une fonction MEX.

Nous comparerons les performances de cette API :

- selon les variations de certains paramètres,
- puis par rapport à MATLAB utilisé seul.

En annexes, nous verrons également un aperçu de plusieurs autres bibliothèques qui pourraient très bien être mises en œuvre dans le cadre de développements de programmes utilisant par exemple la génération de nombres aléatoires ou la manipulation de matrices creuses.

Les annexes comprennent aussi un bref descriptif des diverses améliorations possibles des programmes développés dans ce mémoire ainsi que les étapes de configuration des différents logiciels utilisés.

# I Les cartes graphiques NVIDIA

## I.1 Historique des cartes graphiques NVIDIA

### I.1.1 Les processeurs graphiques antérieurs à CUDA™

Poussés par l'industrie des jeux vidéos, les processeurs graphiques se sont rapidement perfectionnés jusqu'à concurrencer la puissance de calcul des CPU actuels. Ce perfectionnement a permis d'envisager leur utilisation dans de nombreuses applications parallèles à but non graphique.

Les tentatives d'exploiter les processeurs graphiques pour des applications non-graphiques sont apparues dès 2003. Grâce à l'utilisation de langages de traitement graphique de haut niveau tels que DirectX, OpenGL et Cg, divers algorithmes parallèles ont été transférés au processeur graphique. Les problèmes tels que la tarification de stock-options, les requêtes SQL, ou la reconstitution d'IRM ont atteint sur carte graphique de remarquables améliorations de performances. Les premières tentatives d'utilisation des API graphiques pour du calcul à portée générale sont connues sous le nom GPGPU (General-Purpose Processing on Graphics Processing Units).

Bien que le modèle GPGPU ait ouvert la voie à des améliorations très importantes, il comporte néanmoins plusieurs inconvénients.

- Premièrement, il exige du programmeur une connaissance pointue des API graphiques et de l'architecture GPU.
- Deuxièmement, le traitement graphique était très difficile à découper en étapes adaptées aux ressources fixes, accroissant ainsi considérablement la complexité.
- Troisièmement, les caractéristiques de programmation de base telles que les lectures/écritures aléatoires en mémoire n'étaient pas pris en charge, restreignant considérablement le modèle de programmation.
- Et enfin, ces architectures GPU antérieures à CUDA™ :
    - soit ne prenait pas en charge les nombres flottants double précision (64 bits).
    - soit prenait en charge des nombres flottants double précision non normalisés.

Ceci voulait dire que certaines applications scientifiques ne pouvaient pas être exécutées sur le processeur graphique.

### I.1.2 Première architecture unifiée CUDA : l'architecture G80

Pour aborder ces problèmes, deux technologies-clés ont été introduites :

- l'architecture unifiée G80,
- l'environnement de programmation CUDA, une architecture logicielle et matérielle qui permettait de programmer le processeur graphique avec différents langages de programmation de haut niveau.

Ensemble, ces deux technologies représentaient une nouvelle manière d'utiliser le processeur graphique. Au lieu de programmer des unités graphiques dédiées avec des API graphiques, le programmeur pouvait désormais écrire des programmes C avec extensions CUDA et cibler un processeur à fonctionnement parallèle à portée générale. Ce nouveau type de programmation GPU a été dénommée « GPU Computing » - ce qui voulait dire une prise en charge étendue des applications, une prise en charge élargie du langage de programmation, et une claire séparation d'avec les modèles de programmation GPGPU.

La carte graphique GeForce 8800 a été le produit qui a donné naissance au nouveau modèle de calcul GPU. Lancée sur le marché en novembre 2006, la GeForce 8800, basée sur le processeur graphique G80, a apporté plusieurs innovations-clés au calcul GPU.

Le processeur G80 de la GeForce 8800 était le premier processeur graphique à prendre en charge le langage C, permettant aux programmeurs d'utiliser la puissance du processeur graphique sans devoir apprendre un nouveau langage de programmation.

Le G80 a permis de lancer :

- le modèle d'exécution multi-tâches à instruction unique ou SIMT (*single-instruction multiple-thread*). Dans le modèle SIMT, de nombreuses tâches indépendantes exécutent simultanément une instruction unique. Les instructions SIMT ont la capacité de spécifier l'exécution d'une tâche unique car chaque tâche possède son propre compteur d'adresse d'instruction. Cependant, il n'est pas utile de connaître l'architecture SIMT sous CUDA car la plupart du temps, la performance la meilleure s'obtient lorsque de nombreuses tâches exécutent la même instruction.
- la notion de mémoire partagée et de synchronisation de tâches.

Cependant, la virgule flottante double précision n'était pas prise en charge sur ces processeurs.

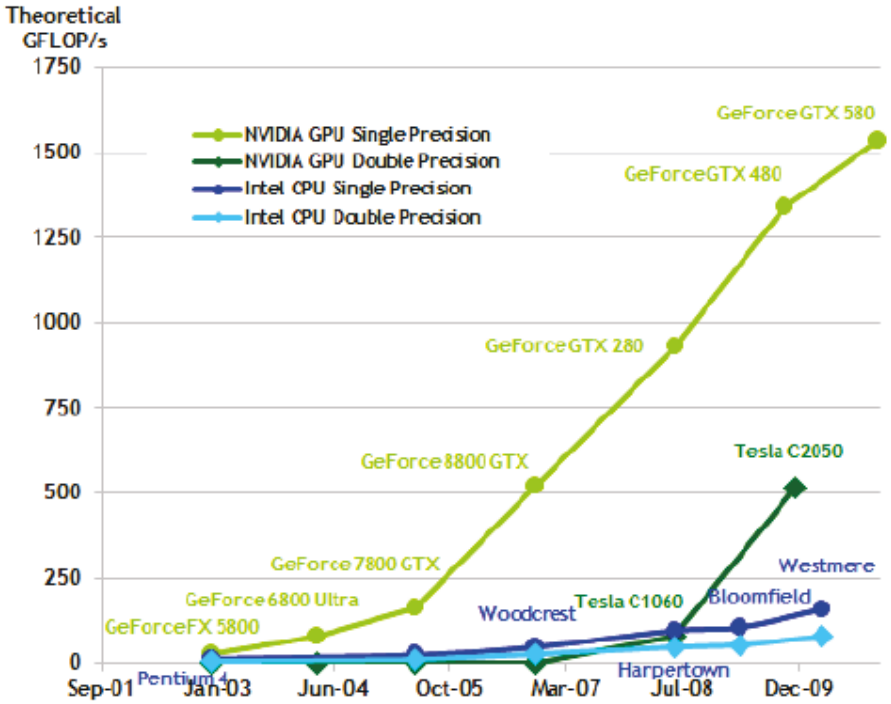



Figure 1 : Comparaison du nombre de GFLOP/s de plusieurs générations de CPU et de cartes graphiques [5].

Sur la Figure 1, on peut comparer l'évolution du nombre de GFLOPS théoriques :

- des cartes graphiques NVIDIA prenant en charge l'utilisation de nombres flottants
  - double précision.
  - simple précision.
- des CPU prenant en charge l'utilisation de nombres flottants
  - double précision.
  - simple précision.

La génération de cartes graphiques NVIDIA contenant un processeur dont l'architecture est :

- antérieure à la technologie CUDA est représentée par les cartes FX5800, 6800 Ultra et 7800 GTX.
- du même type que celle du G80 est représentée par la carte 8800 GTX.
- GT200 est représentée par la carte GTX 280.
- GF100 est représentée par la carte GTX 480.

### I.1.3 Deuxième architecture unifiée CUDA : l'architecture GT200

En juin 2008, NVIDIA a lancé une révision majeure de l'architecture G80. L'architecture unifiée de seconde génération – la GT200 (lancée pour la première fois dans les processeurs graphiques des cartes GeForce GTX 280, Quadro FX 5800, et Tesla T10) – a augmenté le nombre de cœurs de traitement (mentionnés par la suite sous le nom de cœurs CUDA) de 128 à 240.

Le nombre de registres (utilisés pour stocker les variables) par multiprocesseur a doublé, permettant d'exécuter un nombre plus important de tâches sur puce à n'importe quel moment.

Le regroupement des accès à la mémoire matérielle (*coalescing*) a été ajouté pour améliorer l'efficacité de l'accès mémoire. La mémoire globale [5] réside sur la mémoire du périphérique et on accède à la mémoire du périphérique par des transferts mémoire d'une de taille de 32, 64 ou 128 octets. Lorsqu'un ensemble de tâches exécute une instruction qui nécessite d'accéder à la mémoire, les accès mémoire (de chaque tâche) sont regroupés en un ou plusieurs transferts mémoire selon :

- la taille des mots accédés par chaque tâche,
- la répartition des adresses mémoire accédées par chaque tâche.

Pour accélérer les accès à la mémoire, il est donc préférable de regrouper (*coalescing*) les données d'un ensemble de tâches dans des emplacements mémoire dont les adresses se suivent.

La prise en charge de la virgule flottante double précision a également été ajoutée pour répondre aux besoins des applications de calcul haute performance ou HPC (*high-performance computing*) ou des applications scientifiques.

### I.1.4 L'architecture Fermi

L'architecture Fermi représente (en 2011) le bond en avant le plus significatif dans l'architecture GPU depuis le processeur G80. G80 représentait la vision initiale de NVIDIA de la manière dont devait fonctionner un processeur parallèle unifié de traitement graphique et de calcul. Le processeur GT200 a étendu la performance et la fonctionnalité du G80. Avec Fermi, tout ce qui a été appris des deux processeurs antérieurs et des applications écrites pour eux a été utilisé. Au moment de poser les fondations de Fermi, une grande quantité de remarques d'utilisateurs sur le calcul GPU depuis le lancement des processeurs G80 et GT200 a été rassemblée.

Outre le passage de 240 à 512 cœurs de traitement, les améliorations se sont concentrées sur les points clés suivants :

- Prise en charge des nombres flottants double précision (64 bits).
- Prise en charge de code correcteur d'erreur ou ECC (*Error Correcting Code*). L'ECC permet aux utilisateurs de calculs GPU de déployer en toute sécurité de grandes quantités de processeurs graphiques dans les installations de centres de données, et aussi de s'assurer que les applications dont les données sont sensibles

comme l'imagerie médicale et la tarification d'options financières soient protégées contre les erreurs de mémoire.

- Une vraie hiérarchie dans la mémoire cache – Certains algorithmes parallèles étaient incapables d'utiliser la mémoire partagée du processeur graphique, et certains utilisateurs réclamaient une vraie architecture de mémoire cache pour les aider.
- Davantage de mémoire partagée – de nombreux programmeurs CUDA réclamaient plus de 16 ko de mémoire partagée pour accélérer leurs applications.

Dans le Tableau 1, on compare plusieurs caractéristiques des principales cartes graphiques (en 2011) prenant en charge l'environnement CUDA.

Tableau 1 : Cartes graphiques NVIDIA prenant en charge CUDA™

| API graphique | Série et année de lancement | GPU | Compute Capability (versions de GPU) | Carte | Type d'architecture de GPU |
|---|---|---|---|---|---|
| Direct3D 10.0 | 8 (2006) | G80 | 1.0 | GeForce 8800GTX/Ultra/GTS, Tesla C/D/S870, FX4/5600, 360M | G80 (Tesla) |
| | | G84, G86, G92, G98 | 1.1 | GeForce 8400GS/GT, 8600GT/GTS, 8800GT/GTS, 9600GT/GSO, 9800GT/GTX/GX2, GTS 250, GT 120/30, FX 4/570, 3/580, 17/18/3700, 4700x2, 1xxM, 32/370M, 3/5/770M, 16/17/27/28/36/37/3800M, NVS420/50 | |
| | 9 (2008) | G92a/b, G94a/b, G96a/b, G98 | | | |
| | 100 (Jan. 2009) | | | | |
| | 200 (2008 / 2009) | G92a/b | | | |
| | | GT200a/b | 1.3 | GeForce GTX 260, GTX 275, GTX 280, GTX 285, GTX 295, Tesla C/M1060, S1070, Quadro CX, FX 3/4/5800 | GT200 (Tesla) |
| | 300 (2009 / 2010) | GT215, GT216, GT218 | 1.2 | GeForce 210, GT 220/40, FX380 LP, 1800M, 370/380M, NVS 2/3100M | |
| Direct3D 11 | 400 (Avr. 2010) | GF100 | 2.0 | GeForce (GF100) GTX 465, GTX 470, GTX 480, Tesla C2050, C2070, S/M2050/70, Quadro Plex 7000 | GF100 (Fermi) |
| | | GF104, GF106, GF108 | 2.1 | GeForce GT 420, GT 430, GT 440, GTS 450, GTX 460, GTX 550 Ti, GTX 560 Ti, 500M, Quadro 600, 2000, 4000, 5000, 6000 | |
| | 500 (2011) | GF106, GF108, GF114, GF116 | | | |
| | | GF110 | 2.0 | GeForce (GF110) GTX570, GTX580, GTX590 | |
| | | GF119 | | GeForce GT520 | |

## I.2 Constitution d'une carte graphique

### I.2.1 Présentation générale d'une carte graphique

La carte graphique est l'un des rares périphériques reconnus par le PC dès l'initialisation de la machine.

Sa fonction générale est de convertir des données numériques brutes en données pouvant être affichées sur un périphérique destiné à cet usage (écran, vidéo projecteur, etc...). Cependant, sa fonction la plus courante est d'envoyer à l'écran des images stockées dans sa mémoire, à une fréquence et dans un format qui dépendent d'une part de l'écran utilisé et du port sur lequel il est branché et d'autre part de sa configuration interne.

La plupart des cartes graphiques offrent également certaines fonctions supplémentaires telles que l'accélération de l'affichage de scènes 3D et de graphiques 2D, la capture vidéo, l'adaptation d'un tuner TV, le décodage MPEG-2/MPEG-4, l'interface Firewire, le crayon optique, la sortie télévision ou la possibilité de connecter plusieurs moniteurs. D'autres cartes graphiques modernes hautes performances sont utilisées à des fins plus exigeantes graphiquement, telles que les jeux vidéo.

Le rôle de la carte graphique ne se limite cependant pas à ces fonctions puisqu'elle décharge de plus en plus le processeur central des calculs complexes 3D.

### I.2.2 Fonctionnement général d'une carte graphique

Dans un souci de concision, nous nous limiterons au fonctionnement des cartes les plus récentes.

Lorsque qu'une instruction d'affichage quitte le CPU pour aller au moniteur, elle passe (voir Figure 2) :

- par le bus de la carte mère pour aller dans le processeur graphique via un bus série.
  Dans la configuration que nous utiliserons, le bus utilisé sera le bus série PCI-Express (PCI-Express x16 Gen 2 pour la carte GTX 470) cadencé par une horloge à 2.5GHz pour un débit de 500 Mo/s par voie donc 8 Go/s pour un connecteur à 16 voies.
- du processeur graphique vers la mémoire vidéo, afin de créer une image de l'écran à cet endroit (données sous forme numérique).

Ensuite, si on utilise :

- un moniteur analogique :
  - l'image vidéo numérique est transmise de la mémoire vidéo vers le Convertisseur Numérique Analogique (*RAMDAC*) pour la convertir sous forme analogique,
  - la vidéo sous forme analogique est ensuite transmise du RAMDAC vers le moniteur via un connecteur DVI-A, DVI-I ou VGA.
- un moniteur numérique, le signal numérique est transmis de la mémoire vidéo vers le moniteur via un connecteur DVI-I ou DVI-D

Le connecteur DVI-I (DVI-Integrated) permet de transmettre un signal analogique ou numérique.

Dans la configuration que nous utiliserons, un moniteur numérique est branché sur un des deux connecteurs DVI-I présents sur la carte GTX 470.

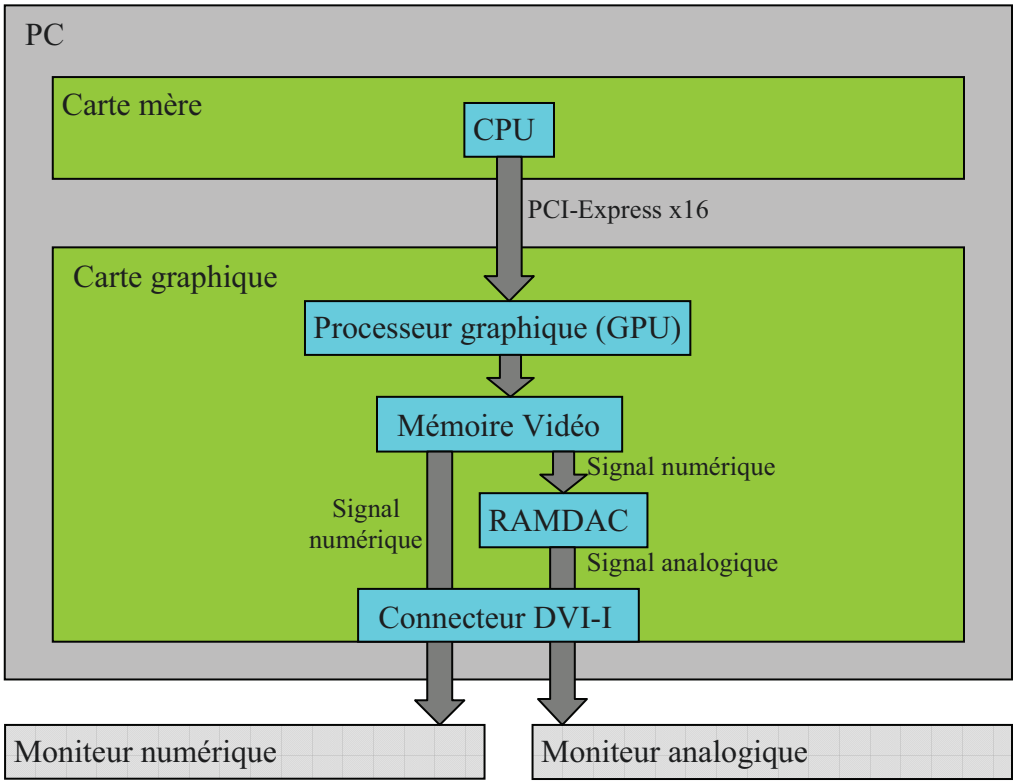



Figure 2 : Etapes de création et/ou de traitement d'une image avant affichage

### I.2.3 Composants d'une carte graphique

Une carte graphique est composée de quatre éléments principaux importants pour le traitement de l'image à afficher :

- le processeur graphique.
- la mémoire vidéo.
- le BIOS vidéo.
- le RAMDAC.

Elle est composée également de plusieurs éléments additionnels :

- la connexion avec la carte mère.
- les entrées-sorties vidéos.
- le système de refroidissement du processeur.

Le processeur graphique

La fonction du processeur graphique ou GPU (Graphical Processing Unit) est de libérer le micro-processeur de la carte mère en prenant en charge les calculs spécifiques à l'affichage et à la coordination des graphismes 3D.

Le processeur est conçu en particulier pour réaliser des calculs en virgule flottante, qui sont fondamentaux pour l'affichage d'images 3D ou 2D. Les principales caractéristiques du GPU sont sa fréquence d'horloge de cœur, qui s'étend typiquement de 250 MHz à 1 GHz et le nombre de pipelines, qui transforment une image 3D caractérisée par des sommets (ou vertex) et des lignes en une image 2D formée de pixels.

Sur la Figure 3, on peut voir la photo d'un des processeurs graphiques précédant CUDA™.


Figure 3 : Le processeur graphique NV43 d'une GeForce 6600 GT

Les processeurs graphiques modernes ont un fonctionnement massivement parallèle, et sont entièrement programmables. Leur puissance de calcul est sans commune mesure avec celle des CPU. En conséquence, ils concurrencent les CPU pour les calculs haute performance.

La mémoire vidéo

La mémoire vidéo conserve les données numériques qui doivent être converties en images par le processeur graphique et les images traitées par le processeur graphique avant leur affichage.

Toutes les cartes graphiques supportent deux méthodes d'accès à leur mémoire. L'une est utilisée pour recevoir des informations en provenance du reste du système, l'autre est sollicitée pour l'affichage à l'écran. La première méthode est un accès direct conventionnel (RAM) comme pour les mémoires centrales, la deuxième méthode est généralement un accès séquentiel à la zone de mémoire contenant l'information à afficher à l'écran.

La capacité mémoire de la plupart des cartes graphiques en 2011 s'étend de 128Mo à 4 Go. La mémoire vidéo est basée surtout sur de la mémoire DRAM de type DDR, DDR2, GDDR3, GDDR4 et GDDR5.

Tableau 2 : Fréquence d'horloge et bande passante selon les types de mémoire

| Type | Fréquence d'horloge mémoire (MHz) | Bande passante (GB/s) |
|---|---|---|
| DDR | 166 - 950 | 1.2 - 30.4 |
| DDR2 | 533 - 1000 | 8.5 - 16 |
| GDDR3 | 700 - 2400 | 5.6 - 156.6 |
| GDDR4 | 2000 - 3600 | 128 - 200 |
| GDDR5 | 900 - 5600 | 130 - 230 |

Le BIOS vidéo

Le BIOS vidéo est à la carte graphique ce que le BIOS est à la carte mère. C'est un petit programme de base enregistré dans une mémoire ROM et qui contient certaines informations sur la carte graphique et dont une des fonctions est de démarrer la carte graphique. Le BIOS vidéo peut contenir des informations sur le cadencement de la

mémoire, sur les vitesses d'opération et les tensions du processeur graphique, sur la RAM et d'autres informations. Le BIOS vidéo gère les opérations de la carte graphique et fournit les instructions qui permettent à l'ordinateur et au logiciel d'interagir avec la carte.

Le RAMDAC

Le RAMDAC (*Random Access Memory Digital-to-Analog Converter*) convertit les images numériques stockées dans la mémoire vidéo (*frame buffer*) de la mémoire vidéo en signaux analogiques à envoyer à un moniteur utilisant des entrées analogiques tel qu'un moniteur à tube cathodique. La fréquence du RAMDAC détermine les taux de rafraîchissement (nombre d'images par seconde en Hz) que la carte graphique peut prendre en charge.

Bien que le RAMDAC soit encore présent dans les cartes graphiques actuelles, il est de moins en moins utilisé, les moniteurs actuels étant numérique pour la plupart. Le RAMDAC ne reste encore utilisé qu'avec un moniteur analogique et une interface DVI-A ou DVI-I.

La connexion avec la carte mère

La connexion de la carte graphique avec la carte mère se fait à l'aide d'un port greffé sur un bus mais dans beaucoup de documentations techniques, on parle plus souvent de bus que de port pour parler de la connexion avec la carte mère.

Au cours des années, plusieurs technologies se sont succédées pour satisfaire les besoins de vitesse de transfert sans cesse croissants des cartes graphiques. On peut les diviser en deux catégories principales :

- Les bus utilisés dans le passé mais actuellement obsolètes : S-100, ISA, NuBus, MCA, EISA, VLB, PCI, UPA, AGP, PCI-X.
- Les bus couramment utilisés actuellement :
  - PCI Express (PCIe) : c'est une interface point à point sortie en 2004 à ne pas confondre avec PCI-X (amélioration des spécifications du bus PCI). Le débit de données peut aller jusqu'à :
    - version 1.0 : 250 Mo/s pour une fréquence de 1,25 GHz
    - version 2.0 : 500 Mo/s pour une fréquence de 2,5 GHz

    Le PCI-Express remplace tous les connecteurs d'extension d'un PC, dont le PCI et l'AGP et est encore (2011) très couramment utilisé dans la plupart des cartes graphiques. On parle de ports PCIe ×1, ×2, ×4, ×8, ×16 et ×32 pour différencier les ports en fonction du nombre de connecteurs de ligne dont ils disposent (respectivement 1, 2, 4, 8, 16 ou 32 lignes maximum). Un port ×32 version 1.0 permet d'atteindre en théorie un débit de 8 Go/s, soit 4 fois le débit des ports AGP.
  - USB : certaines cartes graphiques externes profitent du haut débit (60 Mo/s) qu'offre le bus USB dans sa version 2 mais elles n'arriveront à pleine maturité qu'avec l'USB version 3 (600 Mo/s), permettant d'afficher un nombre d'images par seconde suffisant pour permettre l'affichage de vidéos en mode plein écran.

    La norme USB a un débit de :
    - version 1.0 : 1,5 et 12 Mbits/s
    - version 2.0 : 480 Mbits/s (60 Mo/s)
    - version 3.0: 4,8 Gbits/s (600 Mo/s)

Les interfaces vidéo

Il existe deux types d'interface :

- Les interfaces analogiques. Ces interfaces étant obsolètes ou en voie d'obsolescence, nous ne les détaillerons pas. Les trois principales interfaces analogiques ayant été utilisés sont :
    - L'interface VGA.
    - L'interface vidéo composite (codages vidéo NTSC, SECAM et PAL).
    - L'interface S-vidéo.
- Les interfaces numériques
    - L'interface DVI

      L'interface DVI (Digital Video Interface) permet d'envoyer, aux écrans le supportant, des données numériques. Ceci permet d'éviter des conversions numérique-analogique, puis analogique numériques, inutiles. De plus, la liaison DVI améliore sensiblement la qualité de l'affichage par rapport à la connexion VGA.

      Il existe trois types de prises :
        - le DVI-A (DVI-Analog) qui transmet uniquement le signal analogique;
        - le DVI-D (DVI-Digital) qui transmet uniquement le signal numérique ;
        - le DVI-I (DVI-Integrated) qui transmet (sur des broches séparées) soit le signal numérique du DVI-D, soit le signal analogique du DVI-A.

      Sur la Figure 4, on peut voir les différents types de liaison DVI existants.

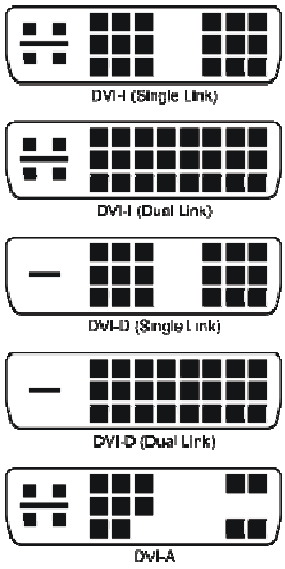



Figure 4 : Types de connecteurs DVI

L'interface DVI existe en mode simple (« single ») ou double (« dual »). Dans le cas des systèmes de liaison double, des broches supplémentaires sont fournies pour la deuxième série de signaux de données.

On choisit le mode de liaison (simple ou double) de la manière suivante :

- Les moniteurs utilisant une horloge inférieure à 165 MHz et moins de 24 bits par pixel doivent utiliser le mode de liaison simple.
- Les moniteurs utilisant une fréquence d'horloge de 165 MHz et/ou plus de 24 bits par pixel doivent utiliser le mode de liaison double.

La liaison double (« dual link ») ne doit pas être confondue avec l'affichage dual (*dual display* également appelé *dual head*), qui décrit les situations pour lesquelles un ordinateur utilise deux moniteurs à la fois pour l'affichage des données.

- L'interface HDMI

Cette interface sortie en 2003 permet de relier la carte à un écran haute définition en transmettant également la partie audio.

Ces signaux sont transmis numériquement et peuvent être cryptés (protection du contenu contre la copie). Elle permet d'interconnecter une source audio/vidéo - tel qu'un lecteur HD DVD ou Blu-ray, un ordinateur, une console de jeu ou un téléviseur HD. Cette interface se base sur l'interface DVI qu'elle étend et supporte aussi bien la vidéo standard que la haute définition.

Il existe trois types de connecteurs HDMI :

- Type A : le plus courant ; il se compose de 19 broches.
- Type B : équivalent du Dual-link DVI mais qui est en partie rendu obsolète par la version 1.3 du HDMI - qui double la bande passante sur un câble de type A.
- Type C : apparu avec la norme 1.3 du HDMI, le type C est une version compacte du type A (avec donc 19 broches) spécialement dédié aux appareils portables tels que caméscopes et appareils photos numériques.
- Type D : ce type de connecteur est défini par la norme HDMI 1.4. Il garde les 19 broches des types A et C mais réduit la taille du connecteur jusqu'à une taille proche de celle d'un connecteur micro-USB.
- Type E : ce type de connecteur est un système de connection utilisé en aéronautique et défini par la norme HDMI 1.4

- L'interface DisplayPort

Cette interface est une interface numérique pour écran mise en place par le consortium VESA (*Video Electronics Standards Association*). Il définit une nouvelle interconnexion numérique audio/vidéo, sans droit ni licence. Celle-ci est d'abord conçue pour relier un PC et ses moniteurs, ou un PC et un système de home cinema.

## I.3 Architecture interne d'une carte graphique NVIDIA

### I.3.1 Présentation

Pour comprendre le fonctionnement parallèle d'une carte graphique, on doit d'abord en connaître la constitution interne et en particulier la répartition de ses différents types de mémoire.

Les cartes graphiques CUDA utilisent plusieurs espaces mémoire, dont les caractéristiques reflètent leurs utilisations. Ces espaces mémoire [6] peuvent être de type global, local, partagé, texture et registres comme il est montré sur la Figure 5.

La Figure 5 montre l'architecture interne générale d'une carte graphique.

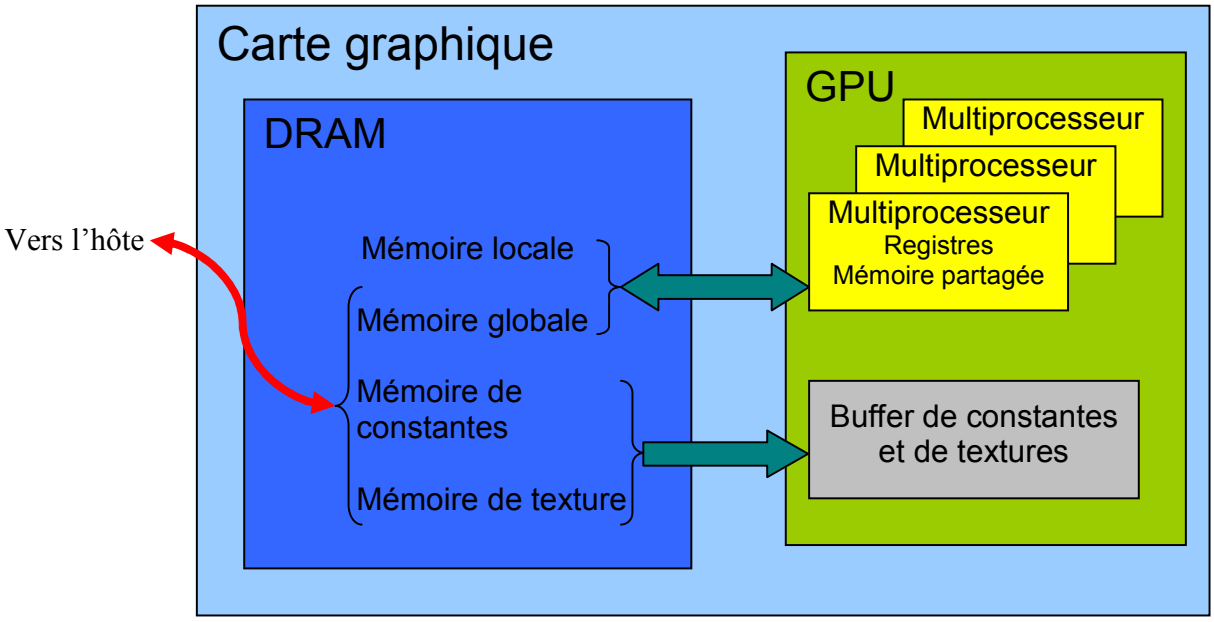



Figure 5 : Répartition des zones mémoire d'une carte graphique

Comme il est montré sur la Figure 5, la mémoire d'une carte graphique se répartit entre la mémoire DRAM et le processeur graphique (GPU).

Parmi ces différents espaces mémoire, les mémoires de type globale et texture sont les plus étendues. Les mémoires de type global, local, et texture ont les temps d'accès les plus importants. Les mémoires de constantes, de registres et la mémoire partagée ont les temps d'accès les plus rapides. La mémoire de constante est plus lente d'accès que la mémoire de registre qui elle-même est plus lente que la mémoire partagée. La mémoire partagée est donc la mémoire la plus rapide d'accès.

Dans le cadre de la programmation CUDA, les parties de la carte graphique les plus importantes à connaître sont le processeur graphique (GPU) et la mémoire DRAM (extérieure au processeur graphique).

Pour bien comprendre l'utilité des différentes zones mémoire, on doit d'abord comprendre la structure d'un processeur graphique, ainsi que la manière dont il gère les calculs à exécuter.

Un processeur graphique est divisé en un nombre variable de multiprocesseurs ou SM (*Streaming Multiprocessor*) eux-mêmes divisés en un nombre variable de cœurs de traitement également appelés processeurs de flux, ou SP (*Streaming Processor*). Un multiprocesseur est conçu pour exécuter des centaines de tâches identiques en même

temps. L'exécution d'un programme CUDA sur carte graphique n'est donc valable que dans les cas où une instruction peut se diviser en de nombreuses tâches identiques. Pour gérer une telle quantité de tâches, un multiprocesseur (*SM*) emploie une architecture à instruction unique et tâches multiples ou SIMT (*Single Instruction Multiple Threads*).

Le langage CUDA-C étend le C en permettant au programmeur de définir une fonction C, appelée *kernel*, qui, lorsqu'elle est appelée, est exécutée N fois en parallèle par N tâches identiques sur N cœurs de traitement distincts. L'ensemble de toutes les tâches à exécuter constitue ce qu'on appelle une grille de tâches. La grille de tâches est elle-même divisée en plusieurs blocs de tâches. Le nombre de blocs par grille et le nombre de tâches par blocs sont configurables par le programmeur dans un programme CUDA mais pour ce faire, une condition doit être respectée : la taille de grille (exprimée en nombre de tâches) doit être un multiple de la taille de bloc.

La grille peut comporter une, deux ou trois dimensions définies par le programmeur :

- 1ère dimension : nombre de lignes de blocs.
- 2ème dimension : nombre de colonnes de blocs.
- 3ème dimension : nombre de blocs en profondeur.

De même, les blocs peuvent comporter une, deux ou trois dimensions définies par le programmeur :

- 1ère dimension : nombre de lignes de tâches.
- 2ème dimension : nombre de colonnes de tâches.
- 3ème dimension : nombre de tâches en profondeur.

Lorsque le kernel est appelé par le code CUDA sur le CPU hôte, le processeur le découpe en blocs de tâches. Cette découpe en blocs est réalisée par une partie du processeur graphique appelée le gestionnaire de tâches.

Les blocs de la grille sont attribués arbitrairement [1] bloc par bloc aux multiprocesseurs disponibles. Un multiprocesseur (*SM*) peut exécuter de 1 à 8 blocs en parallèle. Ces blocs traités en parallèle en permanence sont appelés *blocs résidents*.

Le gestionnaire de tâches connaît la liste des blocs restants à exécuter et attribue de nouveaux blocs aux multiprocesseurs à mesure que l'exécution des blocs précédemment alloués se termine. Les nouveaux blocs attribués deviennent à leur tour blocs résidents. Il y a donc toujours en permanence $n$ blocs en cours d'exécution dans un multiprocesseur (avec $1 \leq n \leq 8$).

Les premiers blocs attribués à un multiprocesseur (*SM*) sont donc tous exécutés immédiatement en parallèle et sont des blocs résidents au moment de leur exécution. Une fois ces blocs exécutés, le multiprocesseur (*SM*) passe au lot de blocs suivant (attribué par le gestionnaire de tâches) qui deviennent à leur tour des blocs résidents. Le nombre de blocs résidents est choisi par le gestionnaire de tâches selon plusieurs paramètres mais ce nombre reste fixe pour une grille de tâches donnée. Le processus de détermination (par le gestionnaire de tâches) du nombre de blocs résidents est décrit plus bas.

Les tâches de chaque bloc résident d'un multiprocesseur (*SM*) sont appelées *tâches résidentes*.

Selon le processeur graphique utilisé, la Figure 6 illustre la façon dont le gestionnaire de tâches attribue des blocs de tâches (qui deviennent résidents) aux multiprocesseurs disponibles.

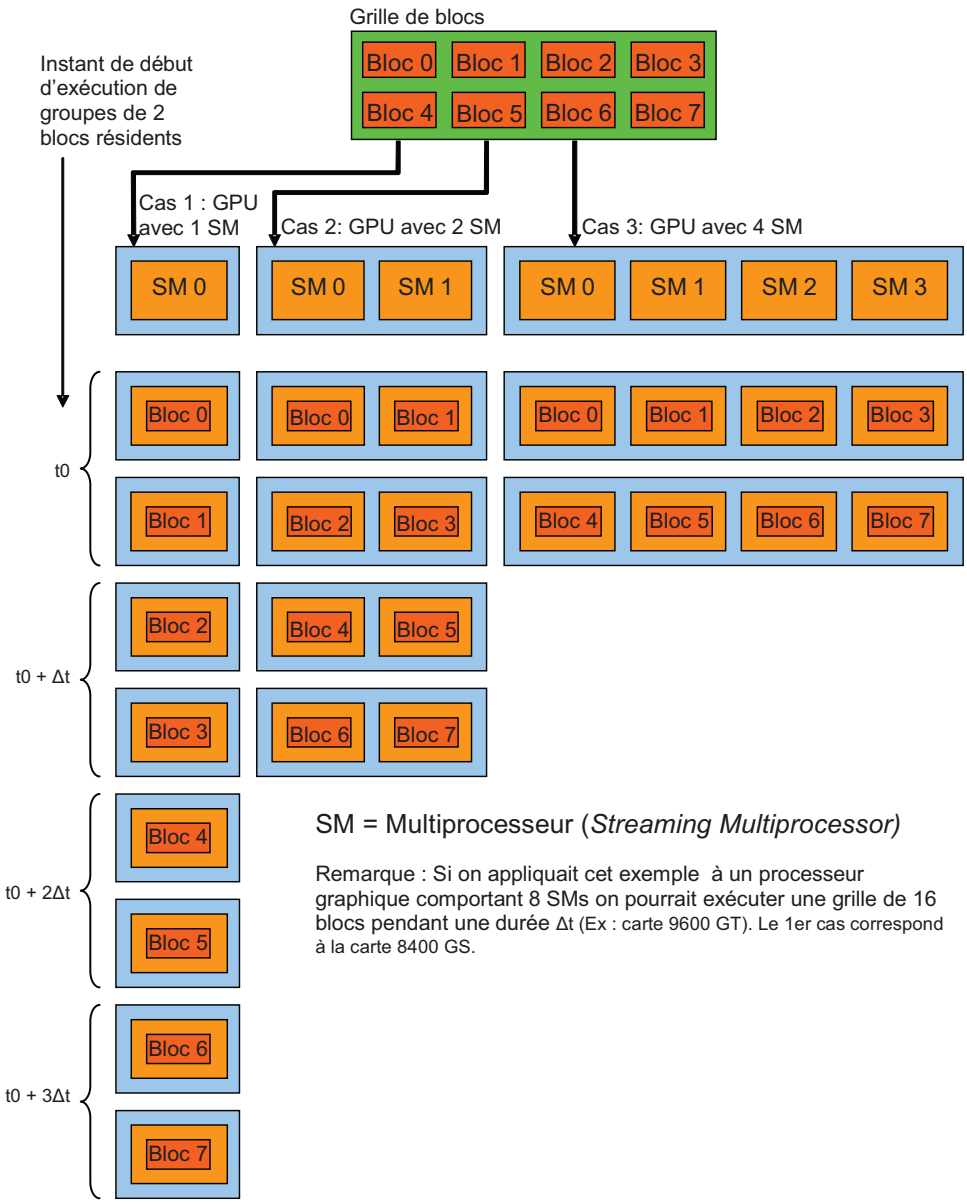



Figure 6 : Répartition des blocs de tâches dans le cas où sont attribués 2 blocs résidents par multiprocesseur (*SM*).

Une fois qu'un ou plusieurs blocs de tâches sont attribués à un multiprocesseur (*SM*), afin d'attribuer les tâches aux cœurs de traitement, le multiprocesseur partitionne les blocs résidents en groupements de 32 tâches appelés *warps*. C'est pourquoi les blocs de tâches doivent comporter un nombre de tâches multiple du nombre de tâches que comporte un warp, c'est-à-dire multiple de 32. A l'intérieur d'un multiprocesseur (*SM*), il existe une unité d'instruction multitâche SIMT qui crée, gère, ordonnance et exécute les warps prélevés dans les blocs résidents. Les 32 tâches d'un warp sont ensuite répartis à raison de :

- 2 tâches par cœur dans le cas de l'architecture Fermi (32 cœurs par multiprocesseur). L'instruction d'un warp est attribuée aux cœurs 0 à 15 et l'instruction du warp suivant est attribuée aux cœurs 16 à 31.
- 4 tâches par cœur (exécutées séquentiellement) dans le cas de l'architecture Tesla (8 cœurs par multiprocesseur).

Quelle que soit l'architecture (G80, GT200, GF100), chaque warp contient toujours 32 tâches. Les warps de tous les blocs résidents d'un multiprocesseur sont appelés *warps résidents*. Un multiprocesseur peut gérer simultanément un nombre fixe de warps résidents. Chaque multiprocesseur d'un GPU peut traiter jusqu'à :

- 24 warps résidents pour l'architecture G80, correspondant à la limite de 768 tâches résidentes maximum.
- 32 warps résidents pour l'architecture GT200, correspondant à la limite de 1024 tâches résidentes maximum.
- 48 warps résidents pour l'architecture GF100, correspondant à la limite de 1536 tâches résidentes maximum.

Un processeur graphique possédant la totalité des caractéristiques de l'architecture GF100 (contenant 16 multiprocesseurs) peut donc traiter un maximum de $1536 * 16 = 24576$ tâches résidentes.

Une autre contrainte à respecter est que chaque bloc ne doit pas contenir plus de :

- 512 tâches dans le cas où l'architecture GPU est de type Tesla (G80 et GT200)
- 1024 tâches dans le cas où l'architecture GPU est de type Fermi (GF100)

Exemple : Supposons qu'on souhaite voir chaque multiprocesseur exécuter 1024 tâches dans un processeur de type GT200. Comme on vient de le voir, jusqu'à 1024 tâches peuvent être attribuées à chaque multiprocesseur. Mais dans cet exemple, ces tâches doivents être réparties en au moins 2 blocs étant donné que chaque bloc ne peut pas contenir plus de 512 tâches.

Selon le choix du programmeur, l'attribution des 1024 tâches peut donc se faire sous une des formes suivantes :

- 2 blocs de 512 tâches.
- 4 blocs de 256 tâches.
- 8 blocs de 128 tâches.

Cette attribution ne peut pas se faire sous les deux formes suivantes :

- 16 blocs de 64 tâches (configuration impossible étant donné que chaque multiprocesseur ne peut pas traiter plus de 8 blocs simultanément).
- 1 bloc de 1024 tâches (configuration impossible car pour l'architecture GT200, un bloc ne doit pas contenir plus de 512 tâches).

On représente une tâche isolée par le symbole de la Figure 7.

Figure 7 : Symbole d'une tâche

Sur la Figure 8, chaque warps est constitué de 32 tâches.

La Figure 8 montre la décomposition des blocs en plusieurs warps. Chaque warp est constitué de 32 tâches dont les numéros d'identification sont consecutifs. Dans cet exemple, trois blocs (bloc 1, bloc 2 et bloc 3) sont attribués à un multiprocesseur. Chacun de ces trois blocs résidents est découpé en warps.

On peut calculer le nombre de warps résidents dans un multiprocesseur pour une taille et un nombre de blocs donnés. Sur la Figure 8, par exemple, si chaque bloc comporte 512

tâches, on peut déterminer que chaque bloc comportera 512 / 32 = 16 warps. Avec 3 blocs par multiprocesseur, il y a donc 48 warps par multiprocesseur, ce qui correspond au maximum possible pour une architecture de type GF100.

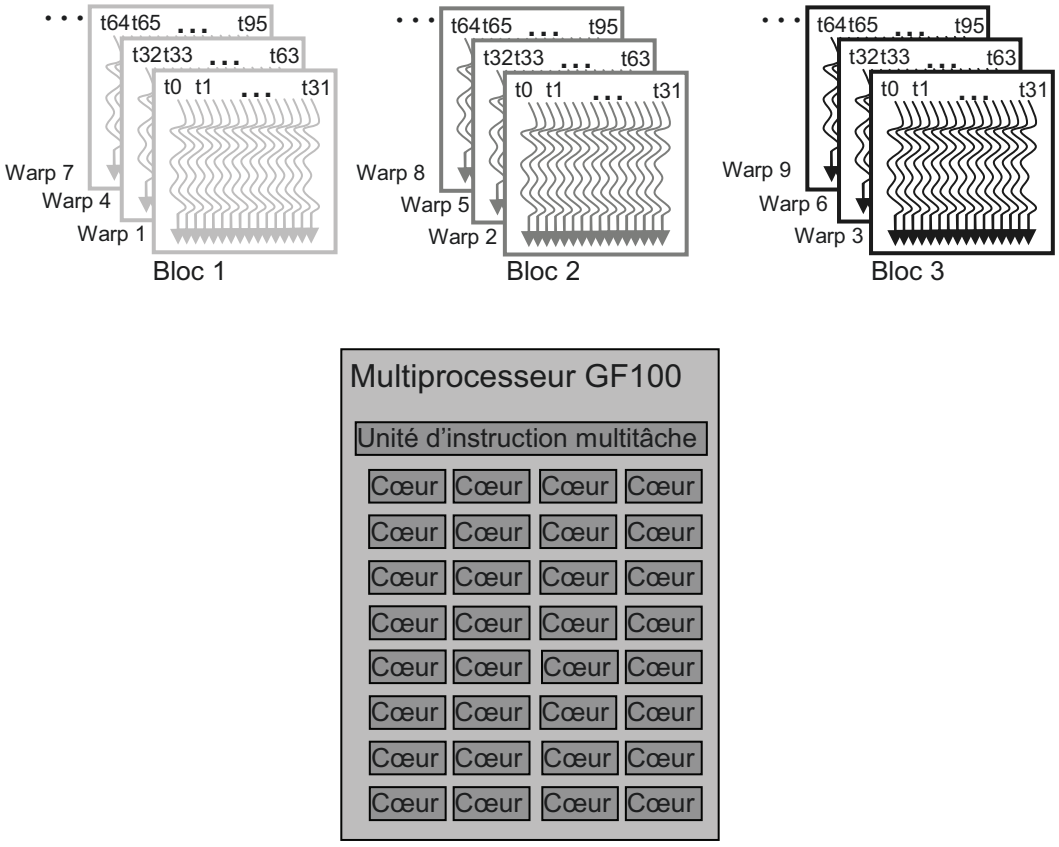



Figure 8 : Attribution de 3 blocs à un multiprocesseur d'une architecture GF100 [1]

Chaque multiprocesseur possède deux planificateurs de warps (*warp scheduler*), chacun des deux est associé à une unité d'instruction multitâche SIMT. Une fois que le gestionnaire de tâches du GPU a attribué 3 blocs par multiprocesseur, chaque bloc résident est découpé en warps par le planificateur de warps (quelque soit le numéro de bloc). Les warps sont ensuite sélectionnés selon l'état des données opérandes, les données opérandes pouvant être disponibles ou non pour l'instruction suivante à exécuter à l'intérieur d'une tâche. Lorsqu'un warp est sélectionné, toutes les tâches qui le composent exécutent la même instruction.

On rappelle que chaque tâche est composée de plusieurs instructions qui peuvent être par exemple :

- Allocation de mémoire partagée.
- Transfert de données opérandes de mémoire globale vers mémoire partagée.
- Calcul.
- Transfert du résultat de mémoire de registres vers mémoire globale.

Certaines de ces instructions nécessitent des données opérandes pour pouvoir être exécutées, d'autres (instructions) non.

Au moment de la sélection des warps à exécuter, deux cas sont possibles :

- l'instruction (à exécuter par les tâches d'un warp) dispose de toutes ses données opérandes (1er cas), alors le warp concerné est sélectionné et envoyé à 16 cœurs pour exécuter l'instruction dont les données opérandes sont disponibles.
- l'instruction (à exécuter par les tâches d'un warp) ne dispose pas de toutes ses données opérandes (2ème cas), alors le warp concerné n'est pas sélectionné car l'instruction à exécuter est encore en attente de ses données opérandes. Alors ce warp est mis en attente jusqu'à ce que son instruction à exécuter dispose de ses données opérandes. Le warp suivants est alors choisi si son instruction à exécuter possède ses données opérandes.

Tout se passe comme si le planificateur de warps répartissait les warps en deux groupes distincts (remis à jour en permanence) :

- l'un composé de warps disposant de leurs données opérandes,
- l'autre composé de warps en attente de leurs données opérandes,

et sélectionnait des warps uniquement dans le groupe disposant de ses données opérandes.

Une fois les warps sélectionnés, les warps à exécuter sont envoyés deux par deux de la manière suivante. Un des warps prêts à être exécutés est envoyé (par l'unité d'instructions multitâches) aux 16 premiers cœurs. En parallèle, l'autre planificateur de warps associé à l'autre unité d'instructions multitâches envoie également un warp prêt à l'exécution aux 16 autres cœurs du multiprocesseur. Par conséquent chaque cœur doit traiter deux tâches parmi les 32 tâches d'un warp (pour l'architecture Fermi).

Dans le cas de processeurs graphiques de type Tesla (G80 ou GT200), étant donné que chaque multiprocesseur ne comporte que 8 cœurs, et un seul planificateur de warps mais qu'un warp est toujours constitué de 32 tâches, chaque cœur traite 4 tâches. 4 cycles d'horloge sont donc nécessaires pour pouvoir exécuter les 32 tâches sur les 8 cœurs.

Dans le cas où le programmeur attribuerait une taille de bloc non multiple de 32, le dernier warp du bloc serait automatiquement rempli (par le multiprocesseur) de tâches supplémentaires fictives (tâches de « bourrage ») jusqu'à atteindre 32 tâches.

La Figure 9 illustre le séquencement de l'exécution des warps de la Figure 8.

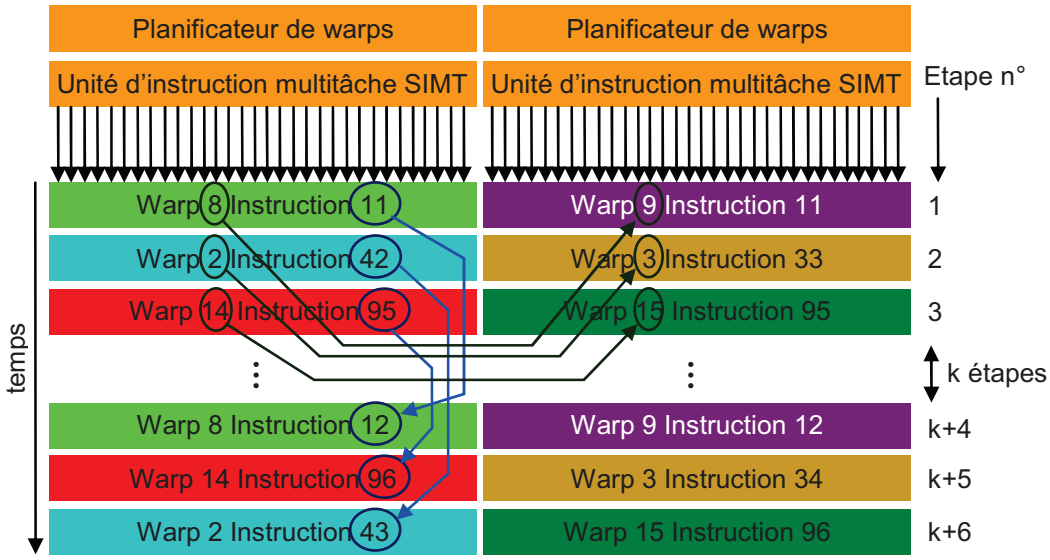



Figure 9 : Ordonnancement des warps et des instructions [7]

Le schéma de la Figure 9 peut s'expliquer en décrivant chacune des étapes :
**Etape n°1** : les 16 premiers cœurs exécutent l'instruction 11 (commune aux 32 tâches) du warp 8 car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 11 du warp 9 car les données opérandes sont disponibles.
**Etape n°2** : les 16 premiers cœurs exécutent l'instruction 42 du warp 2 car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 33 du warp 3 car les données opérandes sont disponibles.
**Etape n°3** : les 16 premiers cœurs exécutent l'instruction 95 du warp 14 car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 95 du warp 15 car les données opérandes sont disponibles.
…
**Etape n°k+4** : les 16 premiers cœurs exécutent l'instruction 12 du warp 8 (*suite de l'étape n°1*) car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 12 du warp 9 (*suite de l'étape n°1*) car les données opérandes sont disponibles.
**Etape n°k+5** : les données nécessaires à l'exécution de l'instruction 43 du warp 2 (*suite de l'étape n°2*) ne sont pas disponibles. Ce warp est donc mis de côté et un autre warp est sélectionné. Alors, à la place, les 16 premiers cœurs exécutent l'instruction 96 du warp 14 (*suite de l'étape n°3*) car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 34 du warp 3 (*suite de l'étape n°2*) car les données opérandes sont disponibles.
**Etape n°k+6** : cette fois-ci, les données opérandes qui manquaient à l'étape k+5 sont disponibles. Donc, les 16 premiers cœurs peuvent exécuter l'instruction 43 du warp 2 (*suite de l'étape n°2*) car les données opérandes sont disponibles.
En parallèle, les 16 autres cœurs exécutent l'instruction 96 du warp 15 (*suite de l'étape n°3*) car les données opérandes sont disponibles.

### I.3.2 Types de mémoire présents sur les cartes graphiques NVIDIA

On appelle :

- hôte (*host*) le CPU du PC contenant la carte graphique
- périphérique de traitement (*device*) la carte graphique

Dans la mémoire DRAM d'une carte graphique :

- la mémoire globale sert à stocker l'ensemble des données envoyées par l'hôte. Elle possède un espace mémoire important mais est lente d'accès. Elle est accessible par la totalité des tâches d'une grille.
- la mémoire locale sert à stocker certaines variables automatiques. Elle est accessible par une seule tâche. Les variables automatiques stockées par le compilateur dans la mémoire locale sont :
    - les tableaux pour lesquels le compilateur ne peut pas savoir s'ils contiennent des nombres constants
    - les structures de grande taille ou les tableaux qui consommeraient trop d'espace de registre.
    - toute variable dans le cas où tous les registres disponibles sont déjà utilisés.

- la mémoire de texture est un type de mémoire utilisé uniquement pour le traitement graphique. Elle est accessible par la totalité des tâches d'un traitement graphique. Elle n'est donc pas utilisée dans notre cas.
- la mémoire de constantes sert à stocker les constantes et est accessible par la totalité des tâches d'une grille de tâches.

Dans le processeur graphique :

- Le buffer de constantes sert à accéder plus rapidement à la mémoire de constantes de la carte graphique.
- Le buffer de texture sert à accéder plus rapidement à la mémoire de texture de la carte graphique. Elle n'est utilisée que pour les traitements graphiques.

Dans chaque multiprocesseur du processeur graphique :

- La mémoire partagée sert à stocker les données à traiter. C'est une mémoire présente en quantité limitée dans chaque multiprocesseur, mais d'accès rapide. Elle est accessible par toutes les tâches d'un bloc.
- La mémoire de registres sert à stocker les variables de chaque tâche. La ressource est partagée entre toutes les tâches résidentes. Une fois le nombre de registres par tâche fixé, une tâche spécifique aura accès à un certains nombre de registres qui ne seront pas accessibles aux autres tâches. La mémoire de registres est limitée à 32768 registres de 32 bits. Si l'ensemble des tâches résidentes utilise plus de 32768 registres, le processeur réduit le nombre de blocs résidents jusqu'à ce que le nombre de registres nécessaires aux tâches résidentes soit inférieur au nombre de registres disponibles.

Les principales caractéristiques des différentes zones mémoires utilisées sur la carte GTX470 sont résumées dans le Tableau 3.

Tableau 3 : Caractéristiques des différentes zones mémoire de la carte GTX 470 [6]

| Mémoire | Emplacement | Portée | Durée de vie | Quantité de mémoire (GTX 470) [5] |
|---|---|---|---|---|
| Registre | Sur GPU | 1 tâche | Tâche | 32768 registres de 32 bits chacun (par multiprocesseur) |
| Locale | Hors GPU | 1 tâche | Tâche | 512 ko par tâche |
| Partagée | Sur GPU | Totalité des tâches d'un bloc de tâches | Bloc de tâches | 48 ko par multiprocesseur |
| Globale | Hors GPU | Totalité des tâches + hôte (GPU) | Durée d'allocation par l'hôte | 1280 Mo de mémoire GDDR5 |
| Constante | Hors GPU | Totalité des tâches + hôte (GPU) | Durée d'allocation par l'hôte | 64 ko par multiprocesseur |
| Texture | Hors GPU | Totalité des tâches + hôte (GPU) | Durée d'allocation par l'hôte | |

D'une manière générale, l'accès à la mémoire GPU est beaucoup plus rapide que l'accès à la mémoire DRAM mais l'espace mémoire GPU est beaucoup plus limité que l'espace mémoire DRAM.

Donc pour optimiser les calculs, on devra faire faire le maximum possible de calculs parallélisables sur la carte graphique, mettre le maximum possible de données en mémoire DRAM et découper le calcul pour envoyer en mémoire partagée des groupements de données au fur et à mesure des calculs. Il faudra éviter de faire faire par

la carte graphique des calculs non parallèlisables pour ne pas surcharger la mémoire DRAM et pour ne pas perdre de temps dans le transfert des données.

### I.3.3 Architectures des processeurs graphiques antérieurs à CUDA™

Dans les toutes premières cartes graphiques, les traitements graphiques étaient relativement simples. Puis les traitements se sont complexifiés et sont devenus un pipeline qui a continué ensuite à se développer. Puis, certaines étapes devenues trop complexes ont été séparées en plusieurs étapes distinctes pour aboutir finalement au pipeline de la Figure 10 représentant les principales étapes d'un pipeline graphique classique (jusqu'à l'API graphique DirectX 9 incluse). Ce type de pipeline est resté d'actualité jusqu'en 2006 [2]. Les processeurs graphiques précédant CUDA™ comportaient un pipeline uniquement graphique et étaient donc encore incapable de traiter des calculs en parallèle.

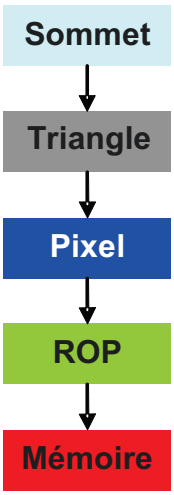



Figure 10 : Pipeline des API graphiques DirectX 9 et antérieurs [2]

Le pipeline de la Figure 10 montre les principales étapes de traitement qu'une image subit avant son affichage. Les étapes de traitement des premiers pipelines graphiques étaient réalisées matériellement puis ils ont été progressivement remplacés par des étapes de traitement programmées appelées *shaders* en anglais. Un *shader* est une suite d'instructions permettant de réaliser des traitements permettant d'obtenir des effets graphiques sur une image tels que par exemple :

- l'absorption et la diffusion de la lumière sur un objet
- la réflexion et la réfraction de la lumière sur un objet
- les ombres d'un objet éclairé

Avant tout traitement par ce pipeline, toute image 3D est découpée en une multitude de surfaces élémentaires de forme triangulaire, chaque surface triangulaire étant définie par ses 3 sommets. Les différentes étapes du pipeline de la Figure 10 sont :

- **Traitement de sommet** : Dans les premières versions de DirectX, l'étape de sommet (*vertex shader*) s'appelait T&L (Transform and Lighting). En effet, cette étape était composée de 3 sous-étapes :
  - Transformée (Transform) : cette étape permettait de convertir des données spatiales depuis un espace virtuel en 3 dimensions vers un espace à 2 dimensions.

- Clipping : Les éléments non visibles (ex: partie de décor masquée par un personnage) de la scène 3D sont éliminés dans cette étape.
- Eclairage (Lighting) : dans cette étape, les objets d'une scène 3D sont éclairés, les effets de la lumière (ombres, réflexions, etc.) sont calculés puis la scène ainsi calculée est envoyée à l'écran.

Cette étape a évolué jusqu'à se réduire (pour DirectX 9) à une étape de traitement de sommet. Le but de cette étape est de transformer chaque position de sommet de l'espace virtuel 3D en coordonnées 2D qui puissent être affichées sur un écran.

- **Traitement de triangle** : Cette étape a pour fonction d'assembler les sommets pour former des primitives (coordonnées d'un ensemble de 3 sommets). Puis les coordonnées de tous les pixels constituant la surface du triangle sont calculés à partir des coordonnées des sommets.
- **Traitement de pixel** : Cette étape a pour but de calculer la couleur de chaque pixel de la surface du triangle en fonction de l'éclairage, des ombres et de la texture appliqués à l'objet. Une texture est une image 2D appliquée à un volume 3D.
- **Traitement ROP** (*Raster Operations*) : Cette étape rassemble l'ensemble des traitements d'image. Les images non visibles sont éliminées dans cette étape comme par exemple lorsqu'une scène dépasse les limites de l'écran ou bien qu'un personnage occulte une partie de la scène. Les effets de transparence plus ou moins prononcée sont traités par cette étape (ex : objets dans l'eau, rideaux, nuages, etc.).

Les pixels traités et finalisés sont ensuite envoyés à la mémoire vidéo avant d'être affichées sur le moniteur.

Cependant, ce type de pipeline présente les principaux inconvénients suivants :

- les données présentes entre les étapes du pipeline sont très peu réutilisées.
- la quantité de ressources matérielles utilisées subit d'importantes variations selon le type de traitement graphique (souvent soit sous-utilisées soit sur-utilisées).
- le jeu d'instruction et les types de données sont limités (manque d'instructions sur nombres entiers et précision en virgule flottante mal définie).
- les résultats présents en milieu de pipeline ne peuvent pas être écrits en mémoire pour être relus en début de pipeline.
- les ressources sont limitées (en terme de registres, textures, instructions par shader, etc.)

Ces nombreux problèmes ont conduit à une architecture totalement nouvelle basée sur le pipeline DirectX 10 : l'architecture GeForce 8800 GPU dite « unifiée ».

### I.3.4 Première architecture unifiée : le processeur G80

#### I.3.4.1 Présentation

L'architecture dite « unifiée » de première génération a été mise en œuvre dans plusieurs processeurs graphiques (tels que le G80, G84, G86, G92, G98). Ces différents processeurs graphiques possédaient la même architecture (appelée architecture Tesla) mais se différenciaient par le nombre de multiprocesseurs utilisables. En fin de fabrication, à l'intérieur de chaque processeur graphique, les multiprocesseurs ne sont pas forcément tous opérationnels. Les processeurs graphiques sont donc triés par le fabricant selon le nombre de multiprocesseurs opérationnels.

Dans cette partie, nous étudierons le processeur qui met en œuvre la totalité de cette architecture : le processeur graphique G80, utilisé pour la première fois sur les

cartes graphiques GeForce 8800. L'architecture G80 était prévue pour fonctionner avec le pipeline de l'API DirectX 10.

Dans les pipelines utilisés dans les architectures précédentes (pipelines des API graphiques DirectX 9 et antérieurs), chaque étape pouvait renfermer jusqu'à 200 sous-étapes. Par son architecture pipeline/shader dite « unifiée », l'organisation du processeur de la GeForce 8800 réduit significativement le nombre d'étapes du pipeline de l'API DirectX 10 (par rapport aux pipelines précédents). L'organisation du flux de traitement n'est plus forcément séquentielle.

Dans les précédentes architectures, les unités de calculs étaient attribuées de manière fixe (un nombre fixe de processeurs par traitement) pour chaque étape de traitement entraînant par exemple (selon l'image à traiter) :

- soit une utilisation de la totalité des unités de calcul dédiées au traitement de pixel associée à une sous-utilisation des unités de calcul dédiées au traitement de sommets. Ex : graphismes contenant peu de figures géométriques.
- soit une sous-utilisation des unités de calcul dédiées au traitement de pixel associée à une utilisation totale des unités de calcul dédiées au traitement de sommets. Ex : dessins contenant beaucoup de figures géométriques.

Avec l'architecture unifiée, les unités de calcul attribuées à chaque étape de traitement sont sélectionnées selon le type de traitement à exécuter : les données nécessaires à une étape du pipeline sont fournies en entrée de plusieurs unités de traitement (*cœurs*) et les sorties sont écrites en registres puis réalimentées en entrée des unités de traitement pour le traitement de l'étape suivante du pipeline.

A chaque boucle, les données sont distribuées dans les unités de traitement puis traitées lors d'une étape, puis elles sont renvoyées et redistribuées de nouveau en entrée des unités de traitement pour subir une autre étape de traitement, et ainsi de suite. Ces opérations (distribution, traitement, rebouclage) sont effectuées jusqu'à ce que le pixel soit transmis au sous-système ROP.

Le modèle de pipeline séquentiel (avant DirectX 10) était appelé « modèle discret » car chaque étape de traitement était bien séparée des autres. Le nouveau modèle de pipeline est appelé « modèle unifié » car tout cœur de traitement peut traiter indifféremment l'une des trois étapes parallélisables qui lui est attribuée : sommet, géométrie ou pixel.

Chacune des trois étapes de traitement (sommet, géométrie, pixel) du pipeline DirectX 10 (représenté sur la Figure 11) peut donc être traitée par des cœurs ayant déjà servi pour d'autres étapes. C'est la partie unifiée.

L'amélioration considérable en termes de flexibilité et de quantité de mémoire de l'architecture G80 permet pour la première fois dans l'histoire des cartes graphiques d'envisager une utilisation autre que graphique des processeurs, en particulier une utilisation pour la réalisation de calculs, délestant ainsi le CPU de tous les calculs parallélisables. C'est pourquoi, désormais, deux modes d'utilisation du GPU sont possibles, définissant ainsi deux pipelines possibles, chacun organisé autour d'une structure commune basée sur un tableau modulable de processeurs ou SPA (*Scalable Processor Array*) [8]. Ces deux pipelines possibles permettent deux modes d'utilisation du GPU :

- un mode graphique (Figure 12)
- un mode calculs (Figure 13)

Nous étudierons donc ces deux pipelines séparément :

- le pipeline graphique pour faire le lien avec le pipeline de l'API DirectX 10.
- le pipeline de calcul comportant des modules communs avec le pipeline graphique.

### I.3.4.2 Le pipeline graphique

Le pipeline graphique de l'architecture G80 est basé sur le pipeline DirectX 10. Nous commencerons donc par un bref aperçu du pipeline DirectX 10 puis nous verrons de quelle manière il est mis en œuvre dans l'architecture G80.

Comme le veut l'architecture unifiée, tout calcul ou toute étape parallélisable est attribuée à des processeurs choisis en fonction du type de tâche à exécuter. Les cœurs de traitement peuvent donc être mis à contribution plusieurs fois afin d'exécuter plusieurs étapes du pipeline de l'API DirectX 10 représentée sur la Figure 11.

Sur la Figure 11, les étapes traitées par des cœurs sont représentées par des « arrondis ». Les étapes représentées par des rectangles ne sont pas exécutées par des cœurs mais par des modules spécialisés du processeur graphique.

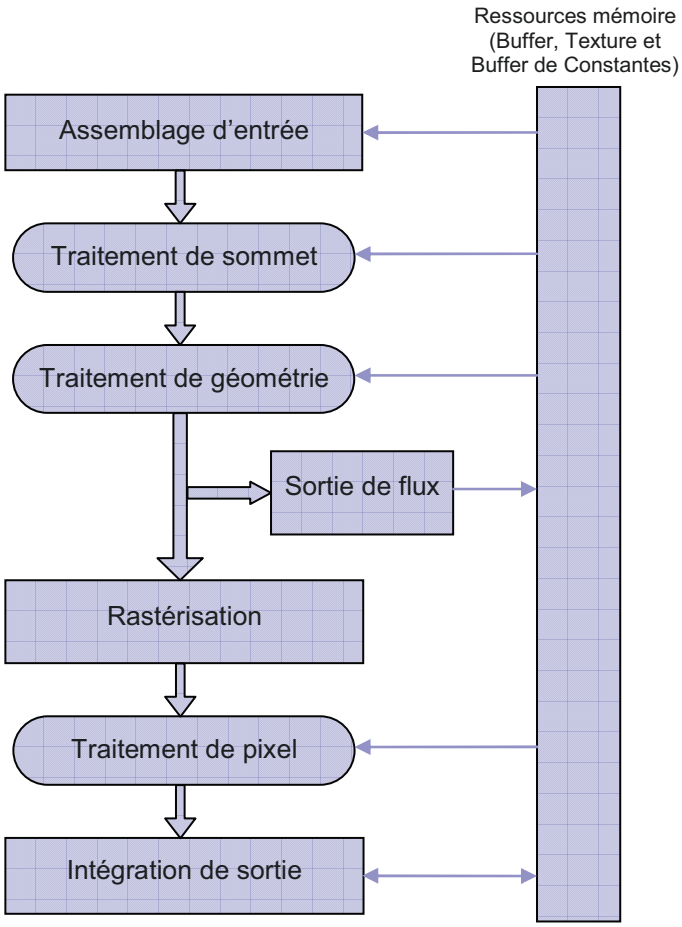



Figure 11 : Pipeline DirectX 10 [2]

Les différentes étapes de la Figure 11 sont décrites ci-dessous :

- **Assemblage d'entrée** : le but cette étape est d'assembler trois sommets (appelés *primitives*) en un triangle élémentaire [9].

- **Traitement de sommet** : le but de cette étape est de transformer les sommets de triangle de l'espace virtuel 3D en coordonnées 2D qui puissent être affichées sur un écran.
- **Traitement de géométrie** : cette étape permet de modifier la géométrie de chaque triangle et éventuellement de créer de nouveaux triangles. Cette étape traite des primitives entières (les trois sommets d'un triangle). Chaque primitive peut inclure les données de sommet des primitives voisines, c'est-à-dire au maximum trois sommets en plus.
- **Sortie de flux** : cette étape envoie de manière continue les données des primitives du pipeline vers la mémoire après le traitement de géométrie. Les données déviées vers la mémoire peuvent être ramenées dans le pipeline comme données d'entrée ou bien lues par le CPU [10].
- **Rastérisation** : les coordonnées de tous les pixels constituant la surface du triangle sont calculées à partir des coordonnées des sommets du triangle [11]. Cette étape sert à convertir les primitives en une image de trame (composée de pixels).
- **Traitement de pixel** : cette étape a pour but de calculer la couleur de chaque pixel de la surface du triangle en fonction de l'éclairage, des ombres et de la texture appliqués à un objet dans une scène. Une texture est une image 2D appliquée à un volume 3D [12].
- **Integration de sortie** : les images non visibles sont éliminées dans cette étape comme par exemple lorsqu'une scène dépasse les limites de l'écran ou bien qu'un personnage occulte une partie de la scène. Les effets de transparence plus ou moins prononcée sont traités par cette étape (ex : objets dans l'eau, rideaux, nuages, etc.). La couleur finale de chaque pixel est déterminée à cette étape.

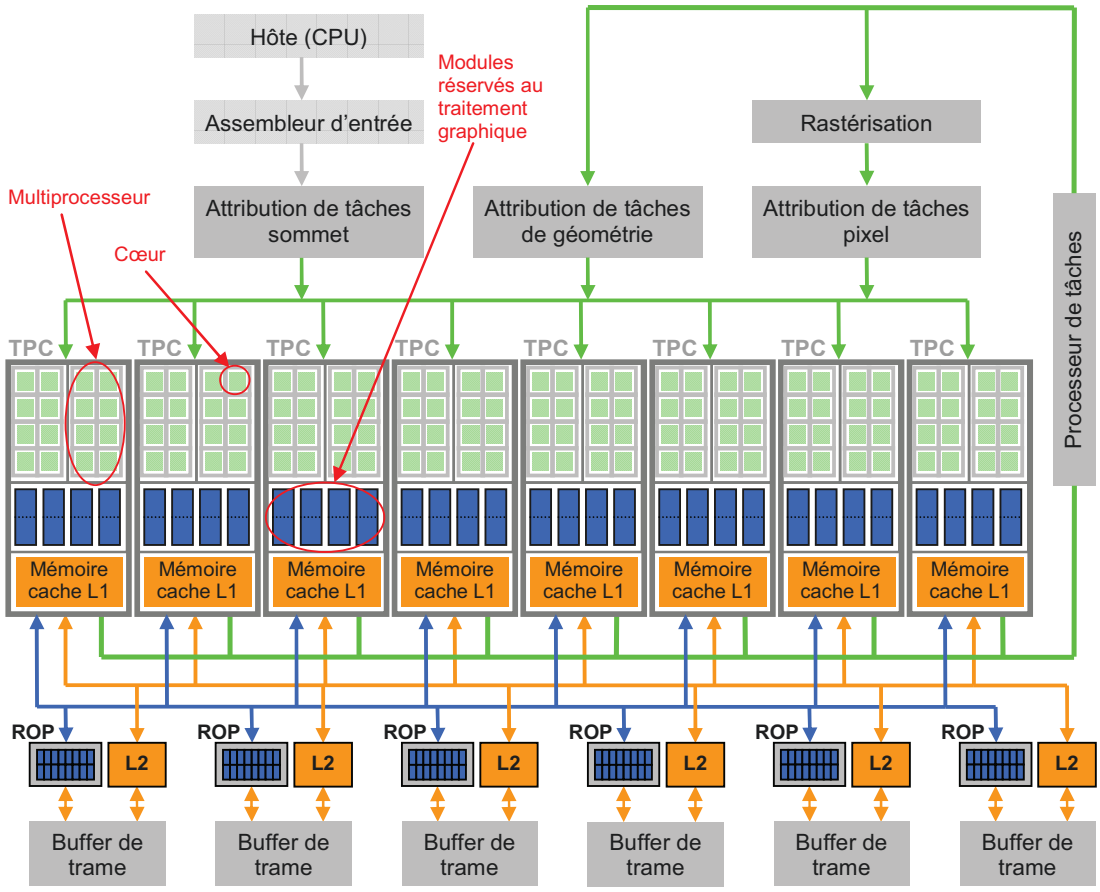


Figure 12 : Pipeline de traitement graphique du processeur G80 [2]

Sur la Figure 12, on peut observer le pipeline de traitement graphique du processeur G80, c'est-à-dire la façon dont est mis en œuvre le pipeline de l'API DirectX 10 dans le processeur G80. Sur ce schéma :

- on retrouve les étapes du pipeline DirectX 10 qui sont exécutées en dehors des cœurs dans des modules spécialisés (*Assemblage d'entrée*, *Rastérisation*)
- on ne retrouve pas les autres étapes (*traitement de sommet*, de *géométrie* et de *pixel*) car elles sont exécutées par les cœurs. Elles ne sont donc pas visibles. Seules les modules d'attribution des tâches associées sont visibles.

La correspondance entre le pipeline DirectX 10 et le pipeline graphique du processeur G80 est décrite dans les lignes qui suivent.

- Sur la Figure 12, le pipeline de traitement graphique commence par l'envoi par le CPU des données nécessaires à l'assembleur d'entrée du GPU.
- L'assembleur d'entrée est un module GPU spécialisé qui exécute la première étape de traitement du pipeline DirectX 10 de la Figure 11.
- Puis le traitement de sommet du pipeline DirectX 10 est découpé en tâches qui sont attribuées aux processeurs par l'étape d'attribution de tâches sommet.
- Les processeurs choisis exécutent le traitement de sommet, puis envoient les données traitées au processeur de tâches qui sait à quel niveau du pipeline DirectX 10 se trouve le traitement graphique.
- Le processeur de tâches connaît les différentes étapes à exécuter dans le pipeline DirectX 10 et décide à quel module de traitement seront envoyées les données, selon les étapes déjà exécutées dans la séquence de traitements. Dans le cas où seul le traitement de sommet a été exécuté, le processeur de tâches transmet les données à l'étape d'attribution de tâches de géométrie.
- Le traitement de géométrie est ensuite découpé en tâches qui sont attribuées aux processeurs par l'étape d'attribution de tâches de géométrie.
- Les processeurs choisis exécutent le traitement de géométrie du pipeline DirectX 10 puis renvoient les données traitées au processeur de tâches.
- Le processeur de tâches envoie ensuite les données au module de rastérisation.
- Le module de rastérisation exécute l'étape de rastérisation du pipeline DirectX 10. Puis les données sont envoyées au module d'attribution de tâches pixel.
- Le traitement de pixel est découpé en tâches qui sont attribuées aux processeurs par l'étape d'attribution de tâches pixel.
- Les processeurs choisis exécutent le traitement de pixel du pipeline DirectX 10 puis renvoient les données traitées au module ROP.
- Le module ROP (*Raster Operations*) exécute l'étape d'intégration de sortie du pipeline DirectX 10.
- Les données traitées sont alors envoyées à la mémoire vidéo avant d'être affichées sur le moniteur.

Bien entendu lorsqu'un module de traitement ou lorsque qu'une partie des cœurs fait subir une étape du pipeline DirectX 10 à un lot de données, les autres modules de traitement ou les autres cœurs peuvent très bien traiter d'autres lots de données en parallèle, justifiant ainsi le terme de pipeline.

Le pipeline graphique basé sur le pipeline DirectX 10 est centré sur une architecture dite SPA. L'architecture SPA consiste en un certain nombre de blocs de traitement de :

- texture ou TPC (Texture Processing Cluster) pour le mode traitement graphique.

- tâches ou TPC (Thread Processing Cluster) pour le mode calcul parallèle.

Le nombre de TPC peut aller de 1 à 8. De même, chaque TPC est composé d'un certains nombre de multiprocesseurs de flux ou SM (*Streaming Multiprocessor*) pouvant aller de 1 à 2 (blocs de huit carrés verts sur la Figure 12).

Chaque multiprocesseur contient huit cœurs représentés sur la Figure 12 et sur la Figure 13 par des carrés verts. Les cœurs sont également appelés « *processeur de flux* » ou SP (*Streaming Processor*) ou encore « *processeur de tâche* » (*Thread Processor*). Chaque multiprocesseur contient également des modules de traitement dédiés au traitement graphique.

En début de mémoire, le PC utilisé était équipé d'une carte graphique GeForce 8400 GS.

La carte GeForce 8400 GS existe en 2 versions, une version comportant:

- 16 coeurs (un seul TPC de deux multiprocesseurs) avec horloge de 900 MHz.
- 8 coeurs (un seul TPC comportant un seul multiprocesseur) avec une horloge de 1400 MHz.

C'est la version comportant 1 seul multiprocesseur de 8 coeurs qui était utilisée au début de ce projet. Cependant, après une étude de ses caractéristiques, il s'est avéré que cette carte était particulièrement peu performante par rapport aux cartes existantes. Par conséquent, cette carte a été remplacée par une carte plus performante : la GeForce GTX 470.

La carte GTX 470 est basée sur le processeur GF100 qui utilise 448 des 512 cœurs qu'offre l'architecture Fermi™. L'architecture Fermi sera détaillée plus loin dans ce chapitre.

### I.3.4.3 Le pipeline de calcul

Sur la Figure 13, on constate que les blocs de traitement (TPC) et l'assembleur d'entrée utilisés sont les mêmes que ceux du pipeline graphique. On constate aussi une réduction importante du nombre d'étapes, ce qui est normal étant donné que ce sont des calculs qui sont traités par ce pipeline et non plus des images à afficher.

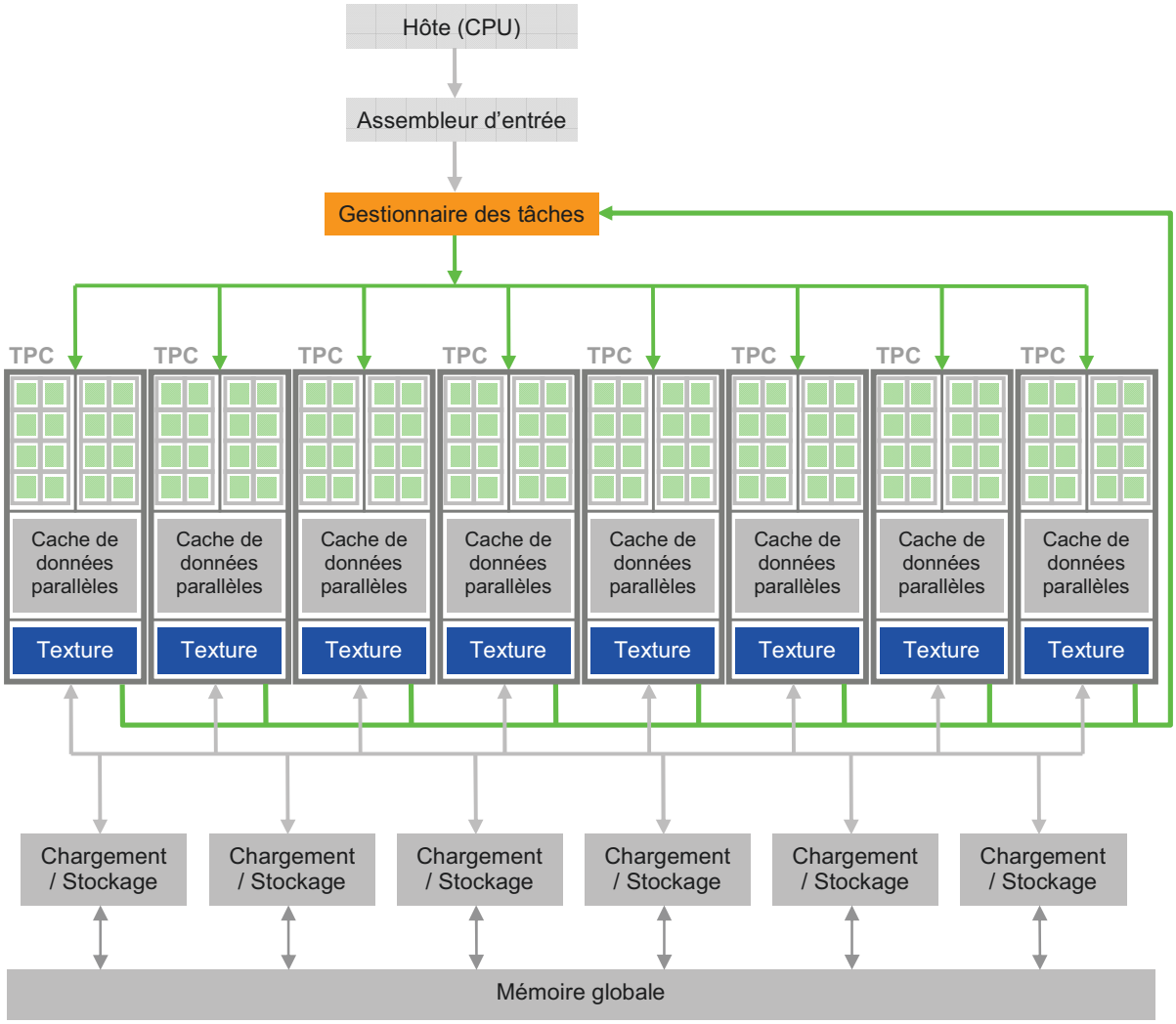



Figure 13 : Pipeline de traitement de calculs du processeur G80 [2] (1ère génération)

### I.3.5 Architecture unifiée seconde génération : le processeur GT200

De même que pour l'architecture des processeurs graphiques de première génération, l'architecture unifiée seconde génération a été mise en œuvre dans plusieurs processeurs graphiques (tels que le G92a/b, GT200a/b, GT215, GT216, GT218). Ces différents processeurs graphiques possédaient la même architecture mais n'utilisaient pas toujours la totalité des possibilités de l'architecture. Dans cette partie, nous étudierons l'architecture du processeur qui utilise la totalité des possibilités de cette nouvelle génération d'architecture : le GT200, qui a été utilisé pour la première fois sur la carte graphique GTX280.

L'architecture des processeurs graphiques NVIDIA de seconde génération est très similaire à celle des processeurs de première génération. Cependant, avec cette 2ème génération on emploie beaucoup plus le terme d'architecture que celui de pipeline car les nombreuses interactions et boucles entre modules ne permettent plus de définir une direction unique de fonctionnement. On n'a plus de flèches indiquant les sens de parcours des modules. Seul le gestionnaire de tâche connaît l'ordre dans lequel les différentes étapes doivent être exécutées et en fonction des étapes en cours, distribue des tâches aux processeurs ou aux modules spécialisés. Les étapes sont :

- les étapes du pipeline DirectX 10 pour le traitement graphique,

- les instructions d'un programme pour le traitement de calculs.

On a donc une architecture réservée au traitement graphique et une architecture réservée au traitement de tâches de calcul. Dans cette partie, nous n'étudierons que l'architecture de calcul, l'architecture graphique n'étant pas utile pour la compréhension de CUDA™.

L'architecture de seconde génération (mise en œuvre dans les processeurs graphiques de la série GeForce GTX 200) est basée sur une architecture SPA étendue, améliorée et réorganisée [8].

La nouvelle architecture SPA de seconde génération des GeForce GTX 280 améliore la performance par rapport aux modèles G80 et G92 de la génération précédente sur deux niveaux :

- Le nombre de multiprocesseurs par TPC passe de deux à trois.
- Le nombre de TPC par puce graphique passe de 8 à 10. Ces deux changements aboutissent à une structure globale de 240 cœurs.

Dans cette nouvelle architecture, on a toujours un gestionnaire de tâche qui planifie les tâches dans les TPC.

Dans cette architecture, la mémoire locale partagée est désormais incluse dans chaque multiprocesseur du TPC alors que dans l'architecture de première génération cette mémoire partagée était commune à deux multiprocesseurs d'un TPC.

Le Tableau 4 présente un résumé des principales différences entre architecture unifiée de première et de seconde génération.

Tableau 4 : Différences d'architecture entre les processeurs des séries GeForce 8/9 et GeForce GTX 200

| GPU correspondant à la série : | Nombre de TPC | Nombre de multiprocesseurs (SM) par TPC | Nombre de cœur par multiprocesseur | Nombre de cœurs total |
|---|---|---|---|---|
| GeForce 8 et 9 | 8 | 2 | 8 | 128 |
| GeForce GTX 200 | 10 | 3 | 8 | 240 |

### I.3.6 Architecture Fermi™: première mise en œuvre avec le processeur GF100

En 2011, le dernier GPU de NVIDIA, nommé GF100, est le premier processeur graphique basé sur l'architecture Fermi™ [13].

L'architecture Fermi™ est mise en œuvre dans plusieurs processeurs graphiques (tels que GF100, GF104, GF106, GF108). Ces différents processeurs graphiques possèdent la même architecture mais n'utilisent pas toujours la totalité des possibilités de l'architecture. C'est-à-dire qu'une partie seulement des processeurs est disponible. Dans cette partie, nous étudierons l'architecture d'un processeur qui utilise la totalité des possibilités de cette nouvelle architecture : le processeur GF100.

Comme pour les architectures précédentes, les processeurs GF100 ont deux modes de fonctionnement possible :

- traitement graphique utilisant l'architecture graphique du processeur,
- traitement de calcul utilisant l'architecture de calcul du processeur.

L'architecture graphique est basée sur le pipeline de l'API graphique DirectX 11.

Nous n'entrerons pas dans les détails de l'architecture graphique du GF100, étant donné que cette architecture n'est pas utilisée pour le traitement de calculs. Nous

étudierons donc uniquement l'architecture de calcul, dont la majeure partie est commune avec l'architecture graphique.

L'architecture de calcul d'un processeur GF100 est basée sur un nombre variable de multiprocesseurs ou SM (*Streaming Multiprocessor*).

Un processeur GF100 met en œuvre seize multiprocesseurs. Chacun des processeurs graphiques basés sur l'architecture du processeur GF100 contient donc différentes configurations de multiprocesseurs. Par exemple, la carte graphique utilisée dans le cadre de ce mémoire (GTX 470) est basée sur un processeur de type GF100 mais ne dispose que de 14 multiprocesseurs sur les 16 possibles.

Chaque multiprocesseur (SM) comporte 32 processeurs CUDA ou cœurs – une concentration quatre fois supérieure à celles des architectures antérieures de multiprocesseurs. Chaque cœur dispose d'une unité arithmétique et logique (ALU) composée de deux unités de calcul, une unité de calcul pour la manipulation des nombres :

- entiers.
- en virgule flottante.

Chacune de ces deux unités de calcul fait partie d'un pipeline.

Dans les architectures précédentes, les GPU utilisaient la norme IEEE 754-1985 pour l'arithmétique virgule flottante. A la différence de ses prédécesseurs, le GF100 met en œuvre la nouvelle norme virgule flottante IEEE 754-2008, disposant de l'instruction multiplication-addition fusionnée ou FMA (*Fused Multiply-Add*) pour l'arithmétique manipulant des nombres pouvant être de type :

- simple précision,
- double précision.

Les instructions MAD et FMA exécutent toutes les deux le produit de deux nombres suivi de l'addition d'un 3$^{ème}$ nombre avec le résultat du produit. La différence entre les deux est que, contrairement à l'instruction MAD, l'instruction FMA ne tronque pas les bits en trop du résultat du produit, ce qui donne une précision bien meilleure.

Dans l'architecture Fermi (GF100), l'unité d'instruction multitâche SIMT évoquée dans le paragraphe I.3.1 se compose de deux ensembles comprenant chacun :

- un planificateur de warps (*warp scheduler*).
- une unité de répartition (*dispatch unit*).

Chaque multiprocesseur comprend deux planificateurs de warps et deux unités de répartition d'instructions, permettant ainsi à deux warps d'être attribués et exécutés simultanément. Ce double planificateur de warps Fermi™ sélectionne deux warps, et attribue une instruction de chaque warp à un groupe de 16 cœurs (unités d'exécution), 16 unités de chargement/stockage, ou quatre SFU [7].

La Figure 14 montre :

- l'architecture interne d'un multiprocesseur Fermi (au centre).
- l'architecture d'un des cœurs d'un multiprocesseur (à gauche).
- un des multiprocesseurs de la Figure 15 (à droite).

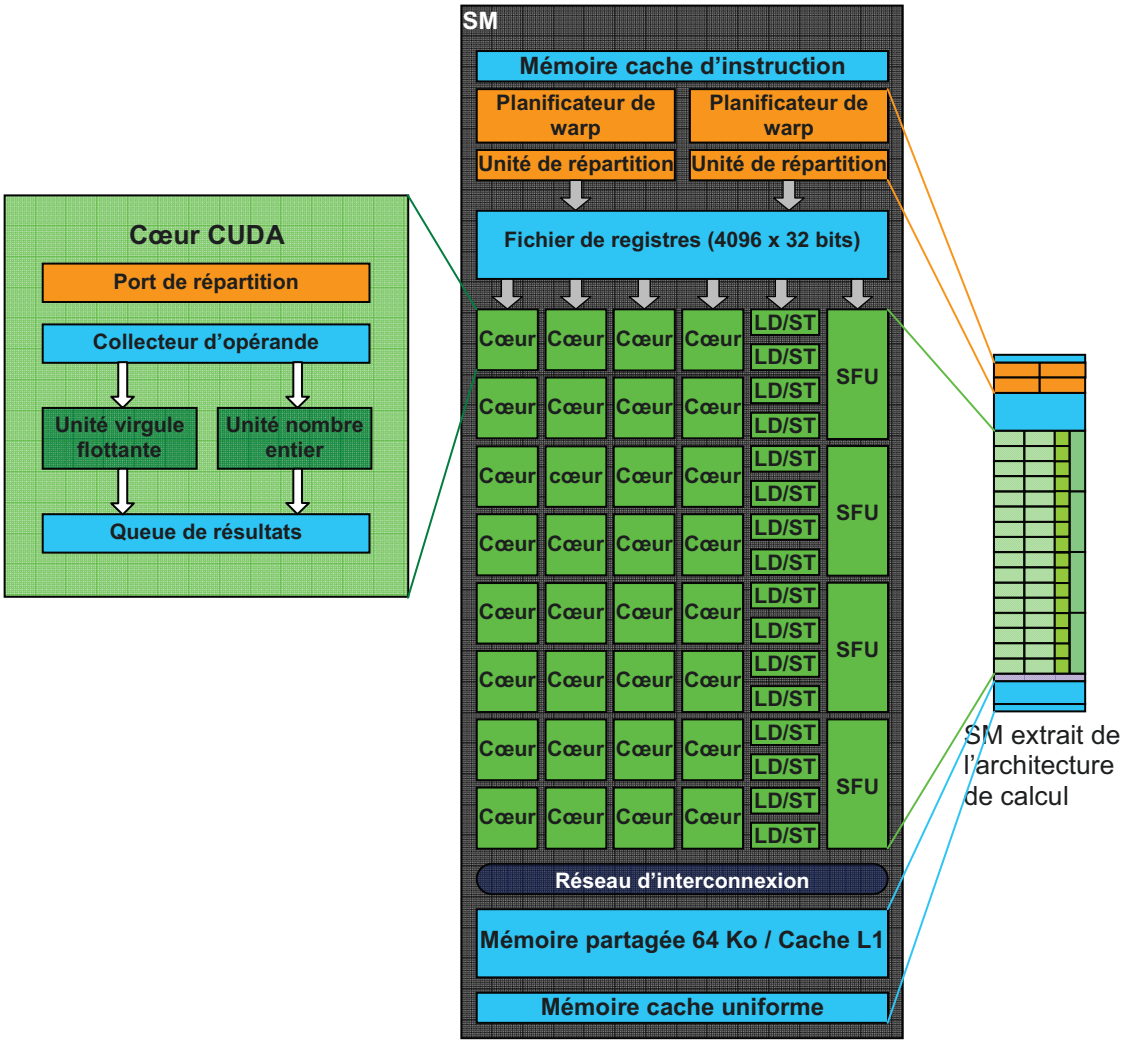



Figure 14 : Architecture d'un multiprocesseur Fermi™

La mémoire partagée :

- permet aux tâches à l'intérieur d'un même bloc de tâches de coopérer,
- facilite une large réutilisation des données sur puce,
- réduit considérablement le trafic hors puce.

Les processeurs G80 et GT200 de l'architecture Tesla possèdent 16 ko de mémoire partagée par multiprocesseur. Dans l'architecture Fermi™, chaque multiprocesseur possède 64 ko de mémoire sur puce qui peut être configurée en :

- 48 ko de mémoire partagée associés avec 16 ko de mémoire cache L1.

ou en :

- 16 ko de mémoire partagée associés avec 48 ko de mémoire cache L1.

La Figure 15 reprend pour chaque multiprocesseur une représentation compacte telle que celle présentée sur la partie droite de la Figure 14. Comme le montre la Figure 15, les multiprocesseurs d'une architecture Fermi™ sont centrés autour d'une mémoire cache commune L2.

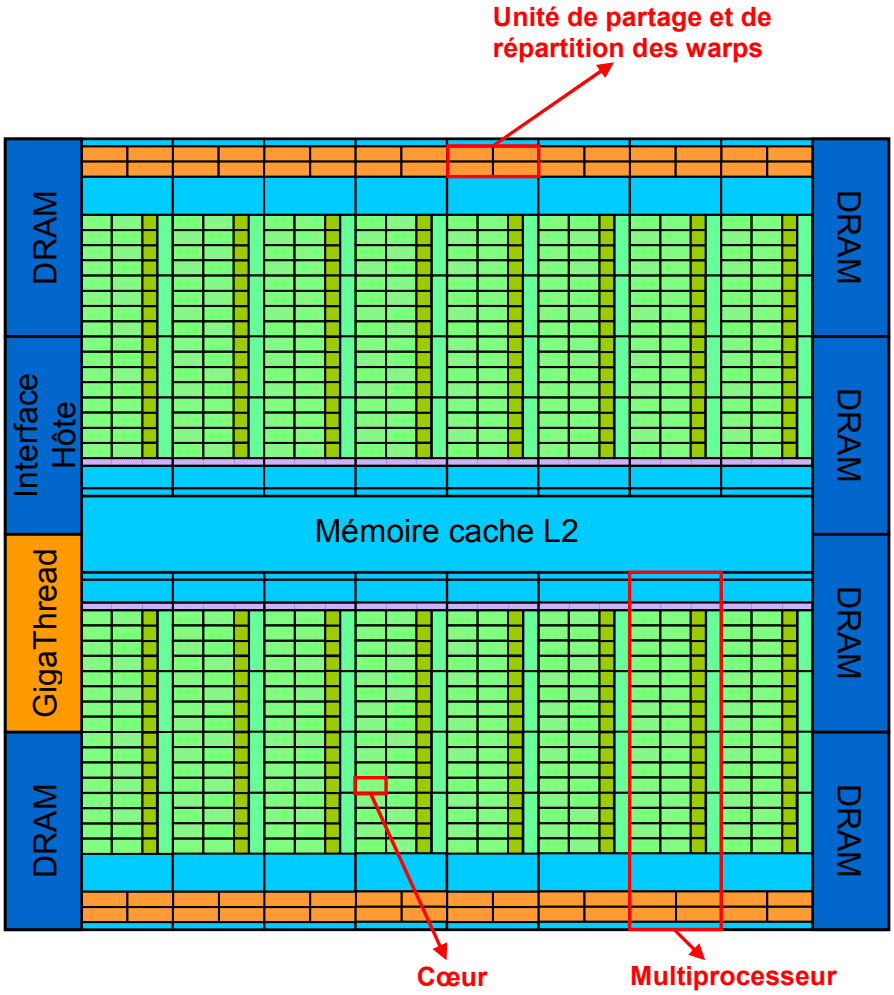



Figure 15 : Architecture Fermi™ en mode calculs

Dans l'architecture Fermi™, la distribution de blocs se fait de la même façon que pour les architectures antérieures à la différence que le gestionnaire de tâches s'appelle ici « moteur GigaThread ». Le processeur possède également 6 Go de mémoire globale de type DRAM GDDR5 répartis en 6 partitions. Chacune de ces partitions est accessible via une interface mémoire de 64 bits. Les six interfaces réalisent donc une interface mémoire totale de 384 bits.

# II Utilisation de l'architecture CUDA™

## II.1 Présentation de l'architecture CUDA™

L'architecture CUDA™ (*Compute Unified Device Architecture*) est une architecture unifiée spécialisée dans l'exécution de calculs. On rappelle qu'une architecture unifiée est une architecture flexible dans laquelle les cœurs d'un processeur graphique peuvent être réutilisés plusieurs fois pour :

- différentes étapes de traitement graphiques,
- l'exécution de calculs non graphiques.

L'architecture CUDA™ est une technologie de type GPGPU (General-Purpose computing on Graphics Processing Units), c'est-à-dire qu'on utilise un processeur graphique (GPU) pour exécuter des calculs généraux habituellement exécutés par le processeur central (CPU). L'environnement CUDA permet de programmer des processeurs graphiques en langage C.

Pour un programmeur CUDA, le système informatique de calcul consiste en un hôte, qui est un CPU traditionnel et un ou plusieurs périphériques de traitement (*device*) qui sont des processeurs à fonctionnement massivement parallèles équipés d'un grand nombre d'unités d'exécutions arithmétiques.

Un programme CUDA consiste en une ou plusieurs phases exécutées soit sur l'hôte (CPU) soit sur un périphérique de traitement tel qu'un processeur graphique. Les phases présentant peu ou pas de parallélisme de données sont mises en œuvre dans le code hôte. Les phases qui présentent une importante quantité de parallélisme sont mises en œuvre dans le code du périphérique de traitement.

Un programme CUDA est un code source englobant à la fois :

- le code hôte,
- le code de périphérique de traitement.

La partie CUDA d'un programme est compilée par le compilateur NVIDIA qui se nomme *nvcc* (*NVIDIA C Compiler*). Ce compilateur sépare les deux types de code pendant le processus de compilation :

- Le code hôte est un code C ANSI ordinaire, il est compilé par les compilateurs standard de l'hôte et s'exécute comme un processus CPU ordinaire.
- Le code de périphérique de traitement est écrit avec des instructions CUDA basées sur le langage C. Ces instructions CUDA sont facilement identifiables car elles comportent le mot « `cuda` » devant l'instruction C correspondante. Par exemple l'instruction CUDA « `cudaMalloc` » comporte le mot « `cuda` » et a un fonctionnement proche de la fonction C « `malloc` ». Le code de périphérique de traitement est typiquement compilé par le compilateur *nvcc* et exécuté sur un périphérique de traitement GPU.

Le kernel (voir §I.3.1) génère typiquement un nombre important de tâches pour exploiter le parallélisme de données.

L'exécution du programme commence par l'exécution sur l'hôte (CPU). Lorsqu'une fonction kernel est appelée, l'exécution se déplace sur un périphérique de traitement (GPU), sur lequel un grand nombre de tâches sont générées pour bénéficier du parallélisme de données. On rappelle que l'ensemble des tâches générées par un kernel lors d'un appel forme ce qu'on appelle une grille (voir §I.3.1).

Lorsque toutes les tâches d'un kernel terminent leur exécution, la grille de tâches correspondante prend fin et l'exécution continue sur l'hôte jusqu'à l'appel d'un autre kernel.

La Figure 16 montre la structure globale d'un programme CUDA-C, ainsi que sa division en grille de tâches.

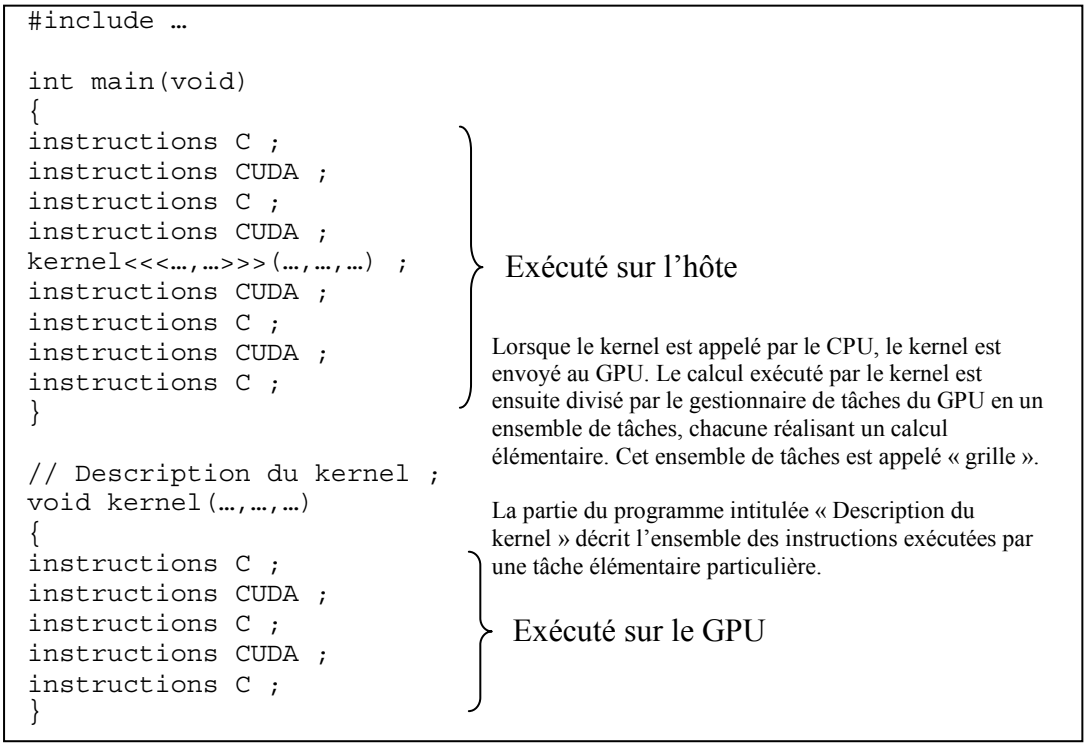



Figure 16 : Structure d'un programme CUDA-C

## II.2 Description d'un programme CUDA de base

L'environnement CUDA étant particulièrement bien adapté à l'exécution de tâches en parallèle, il est donc bien adapté au calcul d'additions ou de multiplications de vecteurs (série de nombres sur une seule dimension) ou de matrices à deux voire trois dimensions. C'est pourquoi l'étude de CUDA™ et de ses bibliothèques de fonctions portera presque toujours sur des matrices ou des vecteurs.

Nous étudierons donc, dans un premier temps, le calcul le plus simple à réaliser dans l'environnement CUDA : une addition de vecteurs contenant par exemple des données de type entier (`int`). On prend des données entières pour garder ce programme de base le plus simple possible.

Un programme CUDA-C de base est constitué principalement des parties suivantes :

- déclaration des variables CPU et GPU,
- allocation de mémoire CPU pour le stockage des données opérandes du calcul à exécuter,
- remplissage des zones mémoire allouées,
- allocation de mémoire GPU pour le stockage sur GPU des données opérandes du calcul à exécuter,
- transferts des données opérandes CPU vers GPU,
- exécution du calcul sur la carte graphique,
- transferts du résultat GPU vers CPU,
- libération des zones mémoires allouées sur GPU,
- libération des zones mémoires allouées sur CPU.

Dans l'exemple suivant on se contentera de faire l'addition de deux vecteurs comportant un seul élément chacun. Voici un programme CUDA réduit à sa plus simple expression :

```
#include <stdio.h>
#include <cuda.h>


__global__ void add( int a, int b, int d )
{
d = a + b;
}


int main( void )
{
// Déclaration de la variable de résultat
int res;
// Déclaration de pointeur sur GPU
int *dev_c;
// Allocation mémoire GPU
cudaMalloc( (void**)&dev_c, sizeof(int) ) ;
// Calcul sur GPU
add<<<1,1>>>( 2, 7, dev_c );
// Transfert du résultat GPU vers CPU
cudaMemcpy( &res, dev_c, sizeof(int), cudaMemcpyDeviceToHost );
// Affichage du résultat du calcul
```

```
printf( "2 + 7 = %d\n", res );

// Libération de la mémoire GPU
cudaFree( dev_c );

return 0;

}
```

On déclare d'abord la variable de résultat avec la ligne :

```
int res;
```

Puis on déclare un pointeur qui servira à pointer l'emplacement mémoire (de la carte graphique) contenant le résultat avant son transfert de la carte graphique vers le CPU :

```
int *dev_c;
```

On utilise le préfixe « dev » pour montrer que le pointeur concerne la carte graphique (*device*).

On réserve de la mémoire pour ce pointeur sur la carte graphique :

```
cudaMalloc( (void**)&dev_c, sizeof(int) );
```

Cet appel se comporte de façon similaire à un appel à `malloc()` en C standard, mais le préfixe `cuda` permet de le différencier. Le premier argument de l'appel à la fonction `cudaMalloc` est un pointeur sur le pointeur qui devra contenir l'adresse de la mémoire qui vient d'être allouée. Le second argument est la taille de l'allocation qu'on veut utiliser. Ici la taille allouée se réduit à celle d'un entier puisqu'on travaille sur un vecteur d'une valeur seulement.

Ensuite le kernel est appelé :

```
add<<<1,1>>>( 2, 7, dev_c );
```

Ce kernel est similaire à la fonction :

```
add( 2, 7, dev_c );
```

La différence entre l'appel d'un kernel et l'appel d'une fonction ordinaire est qu'une fonction ordinaire est exécutée une fois seulement à chaque appel alors qu'un kernel est d'abord découpé en plusieurs blocs de tâches, puis les blocs de tâches sont exécutés simultanément en parallèle. L'ensemble de toutes les tâches constitue une grille de tâches.

Les deux nombres contenus entre les crochets obliques définissent la façon dont seront découpés les blocs et les tâches d'un kernel. Le premier nombre représente le nombre de blocs voulus dans la grille et le deuxième nombre représente le nombre de tâches par bloc. On constate donc que la taille et le nombre de blocs sont configurables par le programmeur.

Dans le kernel, ci-dessus, on a un seul bloc d'une seule tâche. Comme, nous n'avons qu'une seule opération à fournir, il n'y a donc pas de parallélisme.

Les arguments dans la parenthèse de la fonction ont la même utilité que dans une fonction C ordinaire. `2` et `7` sont les deux nombres à additionner et `dev_c` est le pointeur sur le résultat. Les données opérandes (`2` et `7`) sont transmises directement du CPU, c'est pourquoi il n'y a pas eu de transfert CPU vers GPU utilisant `cudaMemcpy`. Dans la pratique, plus un calcul comporte de tâches élémentaires, plus l'utilisation de la carte graphique est valable pour les calculs. Par conséquent, dans les calculs sur carte graphique, l'utilisation de `cudaMemcpy` est quasiment systématique pour le transfert des

opérandes sur la carte graphique comme pour le transfert du résultat de la carte graphique vers le CPU.

Une fois le calcul réalisé, on transfère le résultat de la carte graphique vers le CPU avec l'instruction :

```
cudaMemcpy( &res, dev_c, sizeof(int), cudaMemcpyDeviceToHost );
```

Dans cette fonction CUDA :

- l'argument `&res` représente l'adresse de destination sur le CPU, c'est-à-dire l'adresse du résultat sur le CPU.
- l'argument `dev_c` est le pointeur GPU sur la valeur à tranférer.
- l'argument `sizeof(int)` représente la quantité totale d'octets à transférer. Le résultat n'utilisant d'un seul entier, la quantité d'octets à transférer est donc de la taille d'un `int`.
- l'argument `cudaMemcpyDeviceToHost` est une constante prédéfinie [14] de type `cudaMemcpyKind` représentant la direction du transfert à réaliser. Ici on a donc un transfert « périphérique (GPU) vers hôte (CPU) ».

Puis on libère la mémoire GPU à l'aide de la fonction `cudaFree`.

En début de programme est décrite la fonction `add`. Cette fonction a le même fonctionnement que toute fonction C ordinaire. Le seul changement est la présence du spécificateur de déclaration `__global__`. Le qualificateur `__global__` est prédéfini dans CUDA et permet de déclarer que la fonction `add` est un kernel. Une fonction comportant ce qualificateur est exécutée sur la carte graphique mais ne peut être appelée que par l'hôte [5].

Il existe deux autres qualificateurs de fonctions prédéfinis pouvant être utilisés dans un code CUDA :

- `__device__` : exécuté sur la carte et appelé uniquement par la carte graphique,
- `__host__` : exécuté sur l'hôte (le CPU) et appelé uniquement par l'hôte.

## II.3 Intégration d'un programme CUDA dans une fonction MEX

Pour pouvoir transférer des données à partir d'un programme MATLAB vers un programme en C ou en CUDA-C, on doit utiliser dans le programme C ou CUDA-C une fonction MEX (MATLAB EXecutable) qu'il suffira d'appeler à partir d'une instruction d'un programme MATLAB. Pour utiliser une fonction MEX, on remplace l'entête `main` du programme principal par l'entête `mexFunction`. La fonction `mexFunction` (également appelée « routine passerelle ») comporte quatre paramètres utilisés pour transmettre à la fonction CUDA les données venant de MATLAB. Ces quatre paramètres sont déclarés de la manière suivante :

```
int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[]
```

Les paramètres de la `mexFunction` correspondent aux arguments que MATLAB donne à la fonction lors de son appel [15]. Voici une description de ces paramètres :

- `nrhs` : nombre d'arguments opérandes (arguments d'*entrée*) donnés par MATLAB à la fonction.
- `nlhs` : nombre d'arguments résultats (arguments de *sortie*) devant être renvoyés par la fonction à MATLAB.
- `prhs` : pointeur sur les arguments d'entrée donnés par MATLAB à la fonction (prhs est un tableau).
- `plhs` : pointeur sur les arguments de sortie devant être renvoyés par la fonction à MATLAB (plhs est un tableau).

Le langage MATLAB fonctionne avec un type d'objet unique : le tableau MATLAB. Toutes les variables MATLAB, y compris les variables isolées, les vecteurs, les matrices, les chaînes de caractères, les tableaux, les structures et les objets, sont stockées en tant que tableaux MATLAB. En C, le tableau MATLAB est déclaré comme étant de type `mxArray`. La structure `mxArray` contient entre autres :

- le type de données.
- les dimensions du tableau.
- les données associées à ce tableau.
- en cas de données numériques, le type de données : réel ou complexe.
- en cas de matrice creuse : ses indices et ses éléments maximum non-nuls.
- en cas de structure ou d'objet : le nombre de champs et les noms de champ.
- un pointeur sur les parties :
  - réelles (*pr*),
  - imaginaires (*pi*),

  des données.

La variable `prhs` est déclarée `const` [16]. Ce qui signifie que les valeurs passées dans le fichier MEX ne doivent pas être modifiées. Les valeurs présentes dans `plhs` sont invalides au début de l'exécution de la fonction car les données renvoyées à MATLAB (c'est-à-dire le résultat du calcul) ne sont évidemment pas connues en début d'exécution du programme. Les données de type `mxArray` sur lesquelles `prhs` et `plhs` pointent doivent explicitement être créées avant qu'elles ne puissent être utilisées.

Pour pouvoir envoyer des données MATLAB à la fonction CUDA :

- on crée un fichier `.m` sous MATLAB.
- dans le fichier `.m` , on fait un appel au fichier `.dll` (contenant le code C-CUDA et la `mexFunction`) en indiquant son nom sans mentionner son extension `.dll`.

La syntaxe MATLAB d'appel du fichier MEX-CUDA est la suivante :

```
[X] = add(Y,Z)
```

Le nom `add` donné à cette fonction est donc le même que celui correspondant à une fonction C-CUDA qui serait appelée `add.c` et contenant la `mexFunction`.

On constate que la fonction `add` comporte deux arguments d'entrée (`Y` et `Z`) et une variable de sortie (`X`).

Les groupes de lettres `lhs` et `rhs` représentent l'abréviation de *left-hand side* (côté gauche) et de *right-hand side* (côté droit). lhs et rhs font référence à l'écriture habituelle, où l'on trouve :

- à droite, les arguments passés à la fonction,
- à gauche, la valeur retournée par la fonction.

`nlhs` représente donc le nombre variables de retour (côté gauche) et `nrhs` représente le nombre d'arguments (variables ou constantes) passés à la fonction MEX.

Les arguments passés à la `mexFunction` peuvent être représentés par le schéma de la Figure 17 :

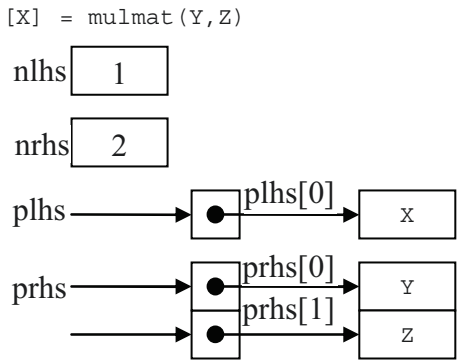



Figure 17 : Description du passage d'arguments MATLAB à une fonction MEX [15]

Sur la Figure 17, `plhs` et `prhs` sont deux tableaux permettant d'accéder aux pointeurs sur les données opérandes (données d'*entrée*) et sur les données résultat (données de *sortie*).

L'entrée est `prhs`, un tableau C de deux éléments (`nrhs` = 2). Le premier élément (`prhs[0]`) est un pointeur sur un tableau de type `mxArray` nommé `Y` et le second élément (`prhs[1]`) est un pointeur sur un tableau de type `mxArray` nommé `Z`.

La sortie est `plhs`, un tableau C de un élément (`nlhs` = 1) dans lequel l'élément unique est un pointeur `null`. Le paramètre `plhs` est un pointeur non alloué car tant que la routine ne s'exécute pas, la sortie `X` n'est pas créée.

La routine passerelle (`mexFunction`) doit créer le tableau de sortie sur lequel elle initialise un pointeur dans `plhs[0]`.

Dans la suite du programme, on s'assure dans un premier temps que le nombre d'arguments d'entrée et de sortie est cohérent avec ce qui est attendu, et on affiche un message dans le cas contraire :

```
if(nrhs!=2) mexErrMsgTxt("Deux entrées demandées.");
if(nlhs!=1) mexErrMsgTxt("Une sortie seulement demandée.");
```

Ensuite pour pouvoir allouer de l'espace mémoire sur l'hôte ou sur la carte graphique, on doit connaître les dimensions des matrices opérandes (matrices d'*entrée*). On utilise donc une fonction permettant d'extraire ces dimensions à partir de `prhs`. On

extrait le nombre de lignes de chaque matrice d'entrée avec la fonction `mxGetM` et le nombre de colonnes avec la fonction `mxGetN` :

```
      mrowsA = (unsigned int)mxGetM(prhs[0]); /* Nb de lignes de la
première matrice */
      ncolsA = (unsigned int)mxGetN(prhs[0]); /* Nb de colonnes de la
première matrice */
      mrowsB = (unsigned int)mxGetM(prhs[1]); /* Nb de lignes de la
deuxième matrice */
      ncolsB = (unsigned int)mxGetN(prhs[1]); /* Nb de colonnes de la
deuxième matrice */
```

On accède aux données via des pointeurs qui font partie de la structure `mxArray`. On récupère la valeur de ces pointeurs au moyen de la fonction `mxGetPr`.

Pour stocker la valeur de ces pointeurs, nous avons pris :

```
double *h_A,*h_B,*h_C;
h_A = mxGetPr(prhs[0]);
h_B = mxGetPr(prhs[1]);
```

Ceci nous permettra dans la suite du programme d'accéder aux données via les pointeurs `h_A` et `h_B`.

Le type `double` a été choisi pour les données car MATLAB ne manipule que des nombres flottants double-précision par défaut et la carte graphique utilisée (GTX470) prend en charge les nombres flottants double-précision. Les données seront toujours considérées de type `double` dans la totalité du programme CUDA-C.

A noter que la carte graphique 8400 GS utilisée en début de projet (voir §I.1.2) ne prenait pas en charge les nombres flottants double-précision[1].

Dans ce programme de multiplication de matrices, nous nommerons :

- matrice `A` la première des deux matrices opérandes.
- `h_A` un pointeur sur la première valeur de la matrice `A`.
- matrice `B` la deuxième des deux matrices opérandes.
- `h_B` un pointeur sur la deuxième valeur de la matrice `B`.

La lettre `h` des pointeurs `h_A` et `h_B` sert à montrer que ce pointeur pointe sur une zone mémoire de l'hôte (le CPU).

Nous avons vu précédemment que le paramètre `plhs` est un pointeur non alloué, on doit donc lui assigner une zone mémoire et un type de données. On utilise l'instruction :

```
plhs[0] = mxCreateDoubleMatrix(mrowsA,ncolsB, mxREAL);
```

Avec cette instruction, le paramètre `plhs` pointe désormais sur une structure `mxArray` de valeurs réelles de type double.

La taille de la matrice résultat doit être adaptée en fonction :

- de ce que produit le calcul,

[1] Pour savoir si une carte graphique prend en charge les nombres flottants double-précision, il suffit de vérifier que la version de carte (*compute capability*) les prend en charge.

- de la taille des matrices opérandes.

C'est au programmeur de fixer correctement la taille de la matrice résultat.

L'argument `mxREAL` est un drapeau de type `mxComplexity`. Le type `mxComplexity` peut être soit `mxREAL` soit `mxCOMPLEX` pour préciser si la donnée qu'on met dans `mxArray` est réelle ou complexe. Dans ce programme, on utilise donc `mxREAL` car on manipule des nombres réels.

Que les données soient réelles ou complexes, la fonction `mxCreateDoubleMatrix` initialise à 0 toutes les valeurs d'un `mxArray`.

Maintenant que le pointeur `plhs` est alloué, on attribue au pointeur hôte `h_C` l'adresse de la première valeur réelle pointée par `plhs[0]` grace à l'instruction :

```
h_C = mxGetPr(plhs[0]);
```

On peut donc désormais utiliser le pointeur hôte `h_C` pointant sur la matrice résultat (matrice de *sortie*). Cette instruction est indispensable pour que la carte graphique puisse transférer le résultat sur l'hôte.

## II.4 Application de CUDA™ à la multiplication de matrices

### II.4.1 Description de la routine passerelle (`mexFunction`)

Une routine passerelle comportant des instructions en CUDA-C doit comporter les étapes suivantes :

- déclaration des variables et des pointeurs CPU (sur les données opérandes),
- initialisation des pointeurs de la routine passerelle (`prhs[0]` et `prhs[1]`),
- déclaration de pointeurs GPU,
- transfert des données CPU vers GPU,
- exécution du calcul sur la carte graphique,
- initialisation du pointeur `plhs[0]` de la routine passerelle,
- transfert du résultat GPU vers CPU au moyen du pointeur `plhs[0]`,
- libération des zones mémoires CPU et GPU.

L'initialisation des pointeurs de la routine passerelle ayant déjà été étudiée (voir §II.3), nous ne reviendrons pas dessus.

Une fois que les données venant de MATLAB ont été retrouvées avec `mxGetPr` et stockées à l'adresse mémoire pointée par `h_A` et `h_B` et que leurs dimensions ont été stockées dans `mrowsA`, `mrowsB`, `ncolsA` et `ncolsB`, on peut commencer à les utiliser. Afin de faciliter le déroulement des instructions, on définit d'abord un type de structure intitulé `Matrix` qui contiendra les dimensions d'une matrice donnée et un pointeur sur les données contenues dans la matrice :

```
typedef struct {
int width;
int height;
int stride;
double* elements;
} Matrix;
```

Une fois le type `Matrix` créé, on peut désormais réserver de la mémoire pour des données de type `Matrix` sur la carte graphique.

On déclare ensuite des pointeurs sur les zones de la mémoire de la carte qui contiendront les deux matrices d'entrée (`d_A` et `d_B`) et la matrice de sortie (`d_C`). La lettre `d` contenue dans les noms de ces pointeurs sert à dire que le pointeur pointe sur une zone mémoire de la carte graphique (**d**evice). Puis on initialise les différents paramètres de la structure.

```
Matrix d_A;
d_A.width = d_A.stride = ncolsA; d_A.height = mrowsA;
```

Dans la structure `d_A`, le paramètre `stride` permet de faire le lien entre le stockage linéaire dans la mémoire et la représentation matricielle. Ce paramètre correspond pour notre cas au nombre de colonnes. Cependant, rien n'interdit de prendre des intervalles inférieurs ou supérieurs au nombre de colonnes d'une matrice si le calcul le nécessite.

On alloue de la mémoire sur la carte graphique aux pointeurs des matrices avec la fonction `cudaMalloc`. Pour allouer de la mémoire on déclare d'abord la variable `size` (de type `size_t`) pour contenir le nombre d'octets à allouer sur la carte. Le type `size_t`

est un type prédéfini du langage C et équivalent à un type `unsigned int` renvoyé par un opérateur `sizeof` [17].

```
size_t size = ncolsA * mrowsA * sizeof(double);
cudaMalloc((void**)&d_A.elements, size);
```

La fonction `cudaMalloc` alloue `size` octets de mémoire sur la carte graphique et renvoie dans `d_A.elements` le pointeur sur la mémoire allouée [14].

On effectue les mêmes opérations pour les pointeurs `d_B` et `d_C`. Le pointeur `d_C` pointant sur la matrice de résultat on utilise comme hauteur (`height`) de matrice la hauteur de la matrice A (c'est-à-dire le nombre de lignes), et comme largeur (`width`) de matrice la largeur (c'est-à-dire le nombre de colonnes) de la matrice B.

Une fois la mémoire allouée sur la carte graphique, on peut transférer les données de l'hôte sur la carte. On transfère donc les deux matrices avec les instructions :

```
cudaMemcpy(d_A.elements, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B.elements, h_B, size, cudaMemcpyHostToDevice);
```

On utilise la fonction `cudaMemcpy` pour transférer les données du CPU vers le GPU.

Dès que les données ont été transférées en mémoire, on peut demander à la carte graphique d'exécuter les calculs voulus. Le détail du calcul parallèle sera décrit plus loin (§II.4.2).

Une fois le calcul exécuté sur la carte graphique, on transfère le résultat sur l'hôte avec la fonction :

```
cudaMemcpy(h_C, d_C.elements, size, cudaMemcpyDeviceToHost);
```

La taille du transfert est définie par la déclaration préalable de `size` :

```
size_t size = ncolsB * mrowsA * sizeof(double);
```

Le résultat est transféré de la zone mémoire GPU pointée par `d_C.elements` vers la zone mémoire CPU pointée par `h_C`.

La direction choisie est : GPU vers CPU (`cudaMemcpyDeviceToHost`).

Une fois le résultat transféré sur l'hôte, il est aussitôt renvoyé à MATLAB puisque `h_C` pointe sur la même zone mémoire que `plhs[0]` (voir §II.3).

Il ne reste ensuite qu'à libérer les zones mémoires réservées sur la carte graphique avec l'instruction :

```
cudaFree(d_A.elements);
```

### II.4.2 Description du kernel

#### II.4.2.1 Appel du kernel

Dans la `mexFunction`, pour lancer l'exécution parallèle, la fonction kernel est simplement appelée comme on l'a vu précédemment (§II.1 et II.2). La fonction kernel est appelée par l'hôte pour être exécutée sur la carte graphique, ce qui est précisé au compilateur par l'emploi de `__global__` devant la déclaration de la fonction :

```
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

Une fois que les données ont été transférées en mémoire, on peut demander à la carte graphique d'exécuter les calculs voulus. Cependant, pour que le kernel puisse être exécuté, on doit lui donner la taille de la grille de tâches (en terme de blocs) et la taille de bloc (en terme de tâches).

Comme il a été vu dans le paragraphe I.3.1, la grille et les blocs peuvent avoir plusieurs dimensions. Ces dimensions sont configurables par le programmeur.

On doit donc définir le nombre de dimensions (1, 2 ou 3) :

- de chaque bloc et la taille de chacune de ces dimensions (nombre de colonnes, de lignes, etc.),
- de la grille et les tailles de chacune de ces dimensions.

Pour un calcul de multiplication de matrices, on souhaite obtenir un résultat sous forme de matrice. Chaque tâche de la grille de tâches calcule un élément de la matrice résultat. Il y a donc autant de tâches dans la grille que d'éléments dans la matrice résultat. Par conséquent, l'idéal est de choisir les mêmes dimensions pour la grille de tâches que pour la matrice résultat. On peut donc choisir les dimensions de grille en fonction des dimensions souhaitées pour la matrice résultat, qui elle-même dépend de la taille des deux matrices opérandes.

La matrice résultat ayant deux dimensions, on choisit donc deux dimensions pour la grille de tâches. Pour les mêmes raisons, on choisira des blocs à deux dimensions.

Il existe une structure CUDA prédéfinie appelée `dim3` et contenant 3 paramètres de type `unsigned int` initialisés à 1 par défaut lorsqu'ils ne sont pas spécifiés par le programmeur. Les paramètres de cette structure sont accessibles par les champs `x`, `y` et `z`.

Pour pouvoir donner les dimensions de grille et de bloc au kernel, on déclare préalablement deux structures de type `dim3` :

- `dimBlock` (pour définir la dimension de bloc).
- `dimGrid` (pour définir la dimension de grille)

`dimBlock` et `dimGrid` comprennent donc chacun 3 paramètres `x`, `y`, `z` correspondant au nombre d'éléments de chacune de leurs 3 dimensions.

Ainsi, une grille comprenant par exemple 5 lignes * 3 colonnes de blocs est définie par :

- `dimGrid.x` = 3
- `dimGrid.y` = 5
- `dimGrid.z` = 1

De même, on définit les dimensions des blocs dans une structure `dim3` appelée `dimBlock` et indiquant le nombre de tâches de chacune des trois dimensions.

Ainsi, un bloc comprenant par exemple 4 lignes * 6 colonnes de tâches est défini par :

- `dimBlock.x` = 6
- `dimBlock.y` = 4
- `dimBlock.z` = 1

Etant donné qu'on n'utilise que deux dimensions dans ce mémoire, on initialisera uniquement les deux premiers paramètres (`x` et `y`), `z` étant initialisé à 1 par défaut.

On a vu dans le paragraphe I.3.1 que la taille de la grille doit être un multiple de la taille de bloc. Pour être encore plus précis, la taille de grille selon la dimension :

- x doit être multiple de la taille de bloc selon la dimension x
- y doit être multiple de la taille de bloc selon la dimension y
- z doit être multiple de la taille de bloc selon la dimension z

Etant donné que le nombre de tâches dans la grille est fixe, le nombre de blocs dans la grille dépend donc directement du nombre de tâches par bloc.

Par exemple, dans le cas d'une grille comportant 16 tâches sur la 1ère dimension et 32 tâches sur la 2ème dimension (Figure 18 et Figure 19) :

- si on choisit des blocs comportant 4 tâches sur la 1ère dimension et 2 tâches sur la 2ème dimension (Figure 18), on aura donc une grille comportant :
  - 16 / 4 = 4 blocs sur la 1ère dimension
  - 32 / 2 = 16 blocs sur la 2ème dimension

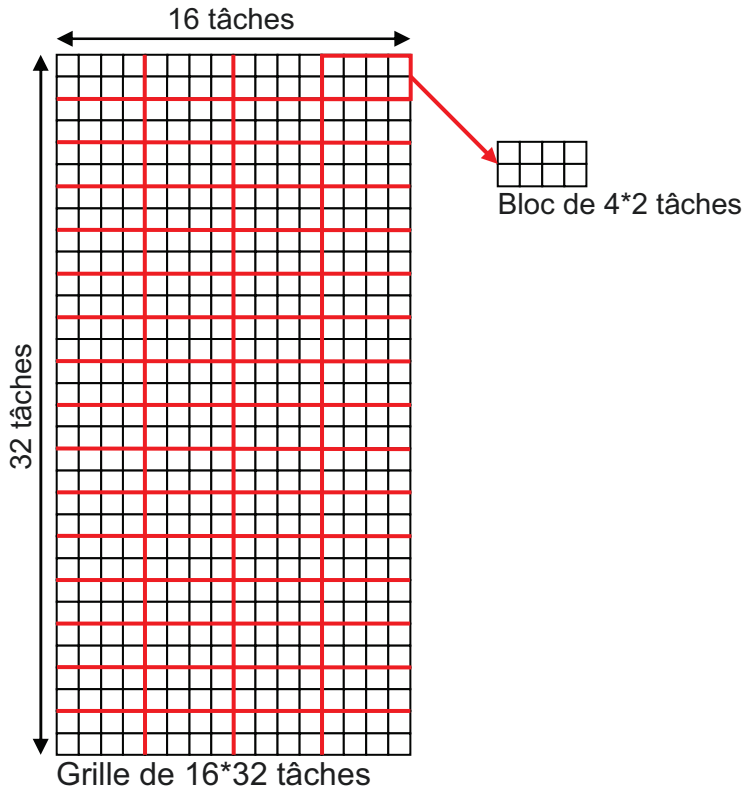



Figure 18 : Détermination de la taille de grille (en terme de blocs) pour des blocs de taille 4*2

- si on choisit des blocs comportant 8 tâches sur la 1ère dimension et 4 tâches sur la 2ème dimension (Figure 19), on aura donc une grille comportant :

- 16 / 8 = 2 blocs sur la 1ère dimension
- 32 / 4 = 8 blocs sur la 2ème dimension

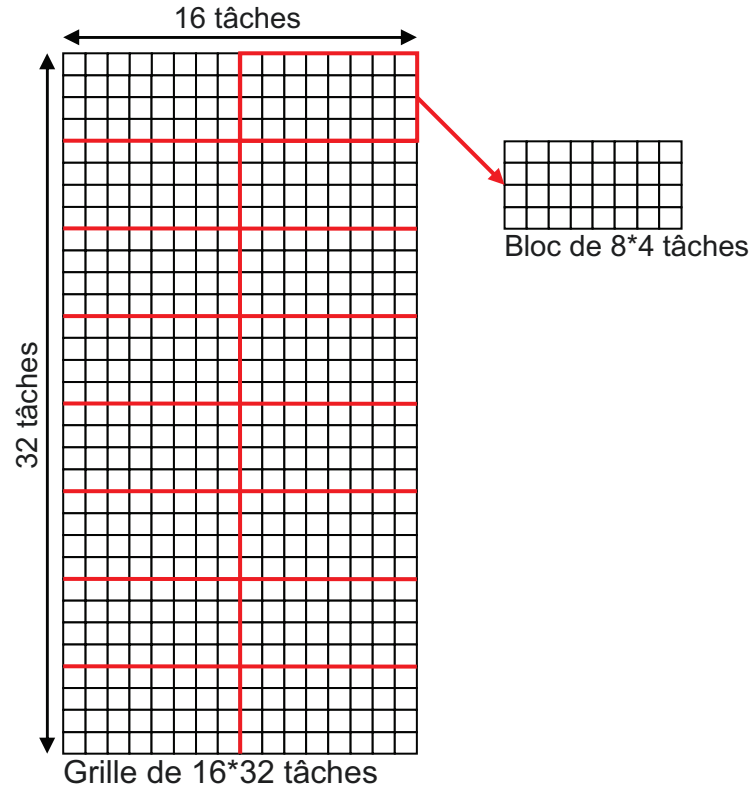



Figure 19 : Détermination de la taille de grille (en terme de blocs) pour des blocs de taille 8*4

On déclare ainsi pour une matrice carrée les dimensions de grille et de bloc de la manière suivante :

```
dim3 dimBlock(BLOCK_SIDE, BLOCK_SIDE);
dim3 dimGrid(ncolsB / dimBlock.x, mrowsA / dimBlock.y);
```

Après l'exécution de ces deux instructions, on a donc :

- `dimBlock.x` = `BLOCK_SIDE` (1er argument)
- `dimBlock.y` = `BLOCK_SIDE` (2ème argument)
- `dimBlock.z` = `1`
- `dimGrid.x = ncolsB / dimBlock.x`
- `dimGrid.x = mrowsA / dimBlock.y`
- `dimGrid.x = 1`

Une fois les dimensions de grille et de bloc établies, il suffit d'appeler le kernel avec l'instruction :

```
MatMulKernel<<<dimGrid, dimBlock>>>(d_B, d_A, d_C);
```

Le kernel `MatMulKernel` est défini comme une fonction C ordinaire. La seule chose qui montre que cette fonction est un kernel est la présence des crochets obliques à l'appel de la fonction. Ceux-ci signalent un calcul multitâche et en précisent les dimensions en blocs (pour la grille) et en tâches (pour chacun des blocs). On ne trouve

ces crochets obliques qu'à l'appel de la fonction. Le détail de l'exécution parallèle sera étudié dans les paragraphes suivants.

### II.4.2.2 Grille et blocs à une dimension

Pour simplifier l'étude de la structuration de la grille en blocs et en tâches, on se limitera dans un premier temps à étudier une matrice ne comportant qu'une seule ligne (une seule dimension : les colonnes).

Pour simplifier les schémas suivants, chaque tâche sera représentée par le symbole de la Figure 20 :

Figure 20 : Tâche isolée

Les tâches sont regroupées dans un bloc comme il est montré sur la Figure 21 :

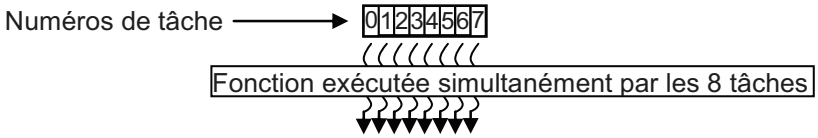



Figure 21 : Bloc de 8 tâches exécutées simultanément

Les tâches à l'intérieur d'un bloc coopèrent entre elles via une zone de mémoire partagée dédiée au bloc concerné. Les tâches de blocs différents ne peuvent pas coopérer via la mémoire partagée.

L'ensemble de plusieurs blocs constitue une grille de tâches comme il est montré sur la Figure 22.

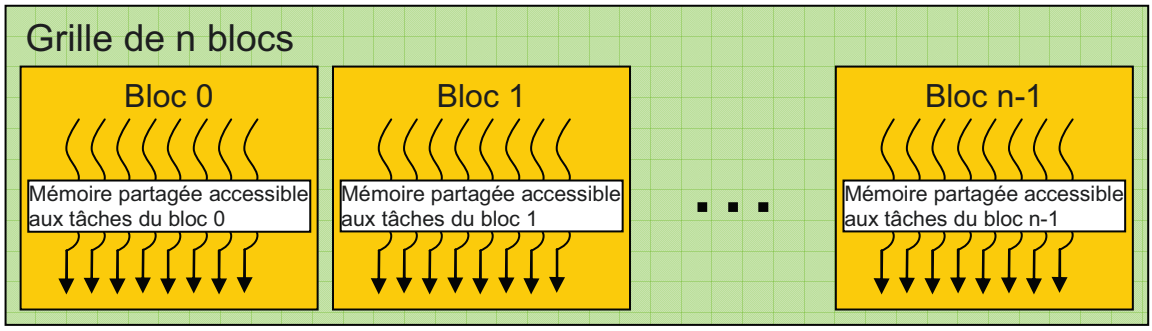



Figure 22 : Grille de tâches découpée en n blocs de tâches

On sait que chaque tâche exécute la même fonction en même temps que les autres tâches. Il existe un moyen clair pour identifier ces tâches. Lors de l'exécution d'une grille de tâches dans le processeur graphique :

- chaque tâche est identifiable à l'intérieur d'un bloc par un numéro d'identification de tâche (qu'on appellera « indice de tâche ») pour chacune des dimensions du bloc concerné,

- chaque bloc est identifiable à l'intérieur de la grille par un numéro d'identification de bloc (qu'on appellera « indice de bloc ») pour chacune des dimensions de la grille.

Chaque tâche utilise à la fois ses propres indices de tâche et l'indice du bloc dans laquelle elle se trouve dans le but de calculer les adresses mémoire :

- des données auxquelles elle accède.
- des variables de la matrice résultat qu'elle met à jour [18].

Pour calculer les zones mémoire auxquelles elle doit accéder, chaque tâche a accès aux variables prédéfinies suivantes mises à jour automatiquement :

- `threadIdx.x` : numéro de tâche à l'intérieur d'un bloc (à partir de 0).
- `blockIdx.x` : numéro de bloc à l'intérieur d'une grille (à partir de 0).
- `blockDim.x` : nombre de tâches par blocs.

Toutes les fonctions possédant le qualificateur `__global__` et `__device__` dans leur déclaration ont accès à quatre variables prédéfinies : `threadIdx, blockIdx, blockDim, gridDim` [18]. Ces variables sont toutes des structures prédéfinies contenant trois paramètres (dimensions) mis à jour automatiquement.

L'environnement CUDA repose sur un fonctionnement SIMT (Single Instruction Multiple Thread), c'est-à-dire que de nombreuses tâches exécutent une instruction unique. Cependant, en pratique, le fonctionnement est très proche d'un fonctionnement SIMD car très souvent sous CUDA, les tâches exécutent la même instruction sur des données différentes. On doit donc avoir un moyen de repérer les données auxquelles une tâche a accès. C'est pourquoi le système d'indice que nous venons de voir est indispensable pour que chaque tâche puisse accéder à la donnée correcte.

La mémoire du périphérique peut être allouée :

- soit en tant que mémoire linéaire,
- soit en tant que tableaux CUDA.

Les tableaux CUDA sont des agencements de la mémoire optimisés pour le traitement graphique [5]. Par conséquent, nous utiliserons la mémoire linéaire pour réaliser les calculs. La mémoire linéaire s'accède via des pointeurs. Elle est typiquement :

- allouée en utilisant en utilisant `cudaMalloc()`,
- libérée en utilisant `cudaFree()`.

Avec ce type de mémoire, le transfert de données entre mémoire hôte et mémoire de périphérique linéaire est typiquement réalisé en utilisant `cudaMemcpy()`.

Les données sont donc toujours stockées en mémoire linéaire sous forme de tableau à une dimension (un pointeur), quelle que soit le type de réprésentation des données manipulées :

- vecteur (1 dimension),
- matrice (2 dimensions),
- matrice de vecteurs (3 dimensions).

Dans le cas d'une grille et de blocs à une dimension, on ne parlera donc que de vecteurs. On a donc une grille de tâches constituée d'une seule ligne de blocs et chaque bloc constitué d'une seule ligne de tâches. Globalement, on traite donc une grille d'une seule ligne de tâches.

Les vecteurs opérandes et le vecteur résultat sont rangés dans des tableaux.

Pour pouvoir calculer un élément du vecteur résultat, chaque tâche de la grille doit pouvoir accéder aux données appropriées dans les tableaux contenant les vecteurs opérandes.

On a vu précédemment, que chaque tâche de la grille calcule un élément de la matrice résultat (ici : du vecteur résultat), il y a donc une tâche par élément de la matrice résultat. C'est pourquoi on utilisera les indices de la grille de tâches pour identifier les éléments de la matrice résultat et en déduire les indices des éléments des matrices opérandes nécessaires au calcul.

Pour accéder à l'élément voulu du tableau contenant le vecteur résultat, la tâche concernée doit connaître l'indice approprié du tableau à une dimension.

Pour connaître l'indice k du tableau vecteur résultat, une tâche a accès :

- au nombre de tâches par bloc noté $N$,
- au numéro d'identification de la tâche dans un bloc, noté $i$,
- au numéro d'identification du bloc dans la grille, noté $j$.

Alors, l'indice $k$ dans le vecteur résultat est donné par : $k = j * N + i$.

La Figure 23 illustre la façon dont il est possible de gérer correctement l'accès au tableau commun, en déterminant pour chaque tâche le bon indice.

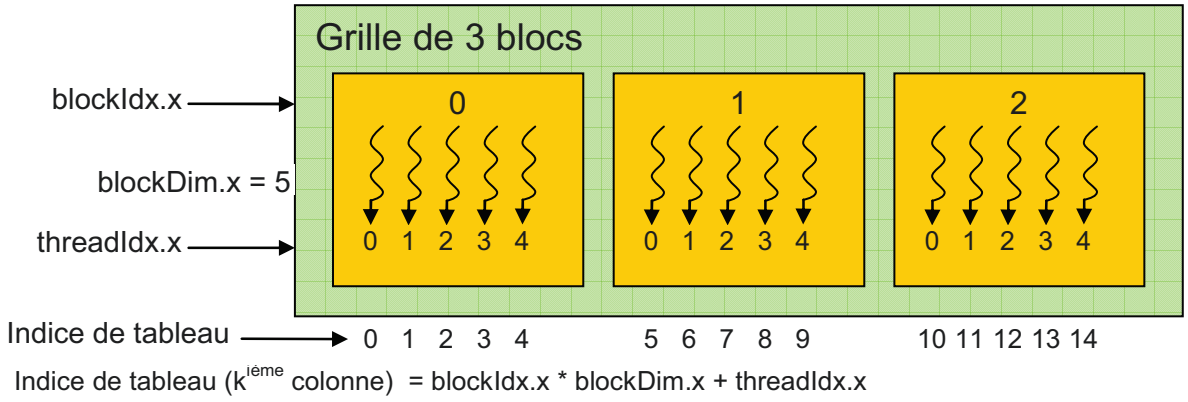



Figure 23 : Grille de tâches découpée en 3 blocs de tâches

Par exemple, sur la Figure 23, la tâche 3 du bloc 1 rangera le résultat dans la colonne $k$ du tableau résultat avec $k = 1 * 5 + 3 = 8$.

### II.4.2.3 Grille et blocs à deux dimensions

Le système d'identification vu précédemment :

- d'une tâche dans une grille à une dimension,
- d'un élément dans le tableau contenant le vecteur résultat (1 dimension),

est généralisable pour des données représentées sur deux dimensions (matrices).

On rappelle que quelles que soient leurs dimensions, les matrices sont toujours stockées dans des tableaux à une dimension.

Dans le cas d'une grille et de blocs à deux dimensions, on ne parlera donc que de matrices. On a donc une grille de tâches constituée de plusieurs colonnes de blocs (1ère dimension) et de plusieurs lignes de blocs (2ème dimension). Chaque bloc est constitué de plusieurs colonnes de tâches et de plusieurs lignes de tâches.

Les matrices opérandes et la matrice résultat sont rangées dans des tableaux.

Pour pouvoir calculer un élément de la matrice résultat, chaque tâche de la grille doit pouvoir accéder aux données appropriées dans les tableaux contenant les matrices opérandes.

On rappelle que chaque tâche de la grille calcule un élément de la matrice résultat. C'est pourquoi on utilisera les indices utilisés dans la grille de tâches pour identifier les éléments concernés dans la matrice résultat.

Etant donné qu'on étudie une grille et des blocs à deux dimensions, on utilisera donc pour se répérer dans la matrice résultat ou dans la grille de tâches, un système similaire au système cartésien ($x$, $y$). La différence est que l'unité y ne sera pas représentée du bas vers le haut (comme dans le système cartésien) mais du haut vers le bas pour pouvoir raisonner plus facilement sur des données stockées en mémoire. De plus les nombres utilisés sont des nombres entiers commencant à 0.

Dans les différentes structures prédéfinies utilisées (`threadIdx`, `blockIdx`, `blockDim`, `gridDim`), on utilisera donc le champ :

- x pour désigner les colonnes,
- y pour désigner les lignes.

Par exemple :

- x = 4 voudra dire « colonne 4 »,
- y = 6 voudra dire « ligne 6 ».

De la même manière, pour la structure `threadIdx` :

- « `threadIdx.x` = 4 » voudra dire « la tâche se situe dans la colonne de tâches n°4 du bloc »,
- « `threadIdx.y` = 6 » voudra dire « la tâche se situe dans la ligne de tâches n°6 du bloc ».

Pour la structure `blockIdx` :

- « `blockIdx.x` = 3 » voudra dire « la tâche se situe dans la colonne de blocs n°3 »,
- « `blockIdx.y` = 5 » voudra dire « la tâche se situe dans la ligne de blocs n°5 ».

Dans la suite du texte, ces numéros d'identification de tâche et de bloc stockés dans les structures `threadIdx` et `blockIdx` seront appelés « indice de tâche » et « indice de bloc » comme précédemment.

Pour la structure `blockDim` :

- « `blockDim.x` = 8 » voudra dire « chaque bloc contient 8 colonnes de tâches »,
- « `blockDim.y` = 4 » voudra dire « chaque bloc contient 4 lignes de tâches ».

Pour la structure `gridDim` :

- « `gridDim.x` = 4 » voudra dire « la grille contient 4 colonnes de blocs »,
- « `gridDim.y` = 16 » voudra dire « la grille contient 16 lignes de blocs ».

Pour accéder à l'élément voulu du tableau contenant la matrice résultat, la tâche concernée doit connaître l'indice approprié (du tableau à une dimension) qui permettra d'accéder au $i^{ème}$ élément du $j^{ème}$ bloc de la matrice résultat.

Etant donné que `threadIdx` représente l'indice d'une tâche dans un bloc et non pas dans la grille, pour pouvoir calculer cet indice de tableau, la tâche doit d'abord calculer l'indice de la tâche dans la grille.

Nous appelerons :

- « `col` » l'indice colonne ($x$) d'une tâche dans la grille,
- « `row` » l'indice ligne ($y$) d'une tâche dans la grille.

Une tâche devra donc calculer :

- son indice « `col` » à partir de :
  - son indice de colonne (`threadIdx.x`) à l'intérieur du bloc dans lequel elle se trouve,
  - l'indice de colonne du bloc (`blockIdx.x`) dans lequel elle se trouve,
  - le nombre de tâches (`blockDim.x`) que comporte une ligne de tâches d'un bloc.
- son indice « `row` » à partir de :
  - son indice de ligne (`threadIdx.y`) à l'intérieur du bloc dans lequel elle se trouve,
  - l'indice de ligne du bloc (`blockIdx.y`) dans lequel elle se trouve,
  - du nombre de tâches (`blockDim.y`) que comporte une colonne de tâches d'un bloc.

L'indice « `col` » se calcule de manière très similaire à l'indice utilisé pour accéder à un tableau à une dimension contenant un vecteur.

On rappelle que pour un vecteur, chaque tâche avait accès :

- au nombre de tâches $N$ : nombre de tâches par bloc.
- à la tâche $i$ : numéro d'identification de tâche dans un bloc.
- au bloc $j$ : numéro d'identification de bloc dans la grille.

L'indice k (dans le tableau contenant le vecteur résultat) de l'élément calculé correspondant à la tâche $i$ du bloc $j$ était donné par : $k = j * N + i$.

Ici pour la dimension « colonne » ($x$), il suffira de remplacer :

- $N$ par `blockDim.x`.
- $i$ par `threadIdx.x`.
- $j$ par `blockIdx.x`.

On aura donc :

`col = blockIdx.x * blockDim.x + threadIdx.x`

De même, `row` se calcule de façon très similaire :

`row = blockIdx.y * blockDim.y + threadIdx.y`

La Figure 24 représente l'organisation des tâches avec :

- une grille de 2 lignes et 3 colonnes de bloc,
- des blocs comportant chacun 3 lignes et 4 colonnes de tâches.

Une tâche particulière est identifiée de façon unique dans le bloc par ses indices (`threadIdx.x` et `threadIdx.y`) et un bloc est identifié également de façon unique dans la grille de tâches par ses indices (`blockIdx.x` et `blockIdx.y`).

Donc la connaissance des indices `threadIdx.x`, `threadIdx.y`, `blockIdx.x` et `blockIdx.y` identifie complètement une tâche parmi toutes celles de la grille. A partir de ces indices, on peut alors déterminer l'emplacement où doit être stocké le résultat du calcul, dans la matrice résultat. Cet emplacement est donné par `row` et `col` tel que :

- `row = blockIdx.y * blockDim.y + threadIdx.y`
- `col = blockIdx.x * blockDim.x + threadIdx.x`

Par exemple, calculons l'indice `row` et `col` de la tâche (2, 1) du bloc (1, 1) sur la Figure 24.

D'abord, on identifie les indices utiles :

- `threadIdx.x = 2`
- `threadIdx.y = 1`
- `blockIdx.x = 1`
- `blockIdx.y = 1`
- `blockDim.x = 4`
- `blockDim.y = 3`

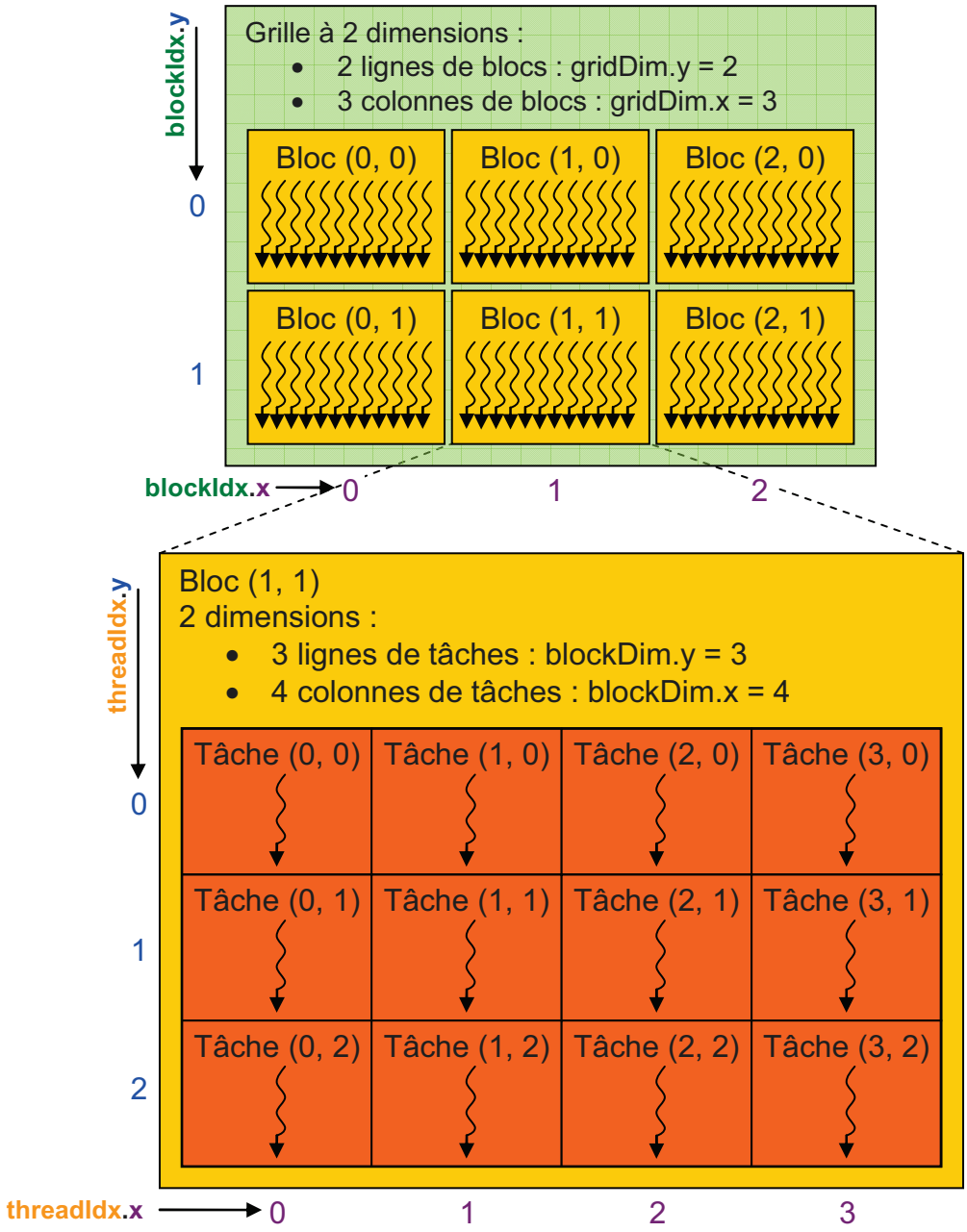



Figure 24 : Description du système d'indices (des blocs et tâches)

On peut alors faire le calcul aisément :

- `row = 1 * 3 + 1 = 4`
- `col = 1 * 4 + 2 = 6`

La tâche en question est donc située :

- sur la ligne de tâches n°4 de la grille,
- sur la colonne de tâches n°6 de la grille.

Une fois que la tâche a calculé ses indices de tâches `col` et `row` dans la grille, elle peut en déduire l'indice de l'élément résultat à calculer dans le tableau à une dimension contenant la matrice résultat.

On rappelle que l'indice de tâche est le même que l'indice de l'élément résultat à calculer dans le tableau à une dimension contenant la matrice résultat. C'est pourquoi, dans l'exemple qui suit, nous ne parlerons plus de tâche mais de donnée.

Avant de calculer cet indice, il faut savoir que CUDA stocke les matrices « *par ligne* » dans un tableau à une dimension. C'est-à-dire que la première ligne de la matrice est d'abord stockée dans le tableau, la deuxième ligne est stockée à la suite de la première, et ainsi de suite jusqu'à la dernière ligne de la matrice.

La Figure 25 illustre le rangement par ligne d'une matrice dans un tableau à une dimension.

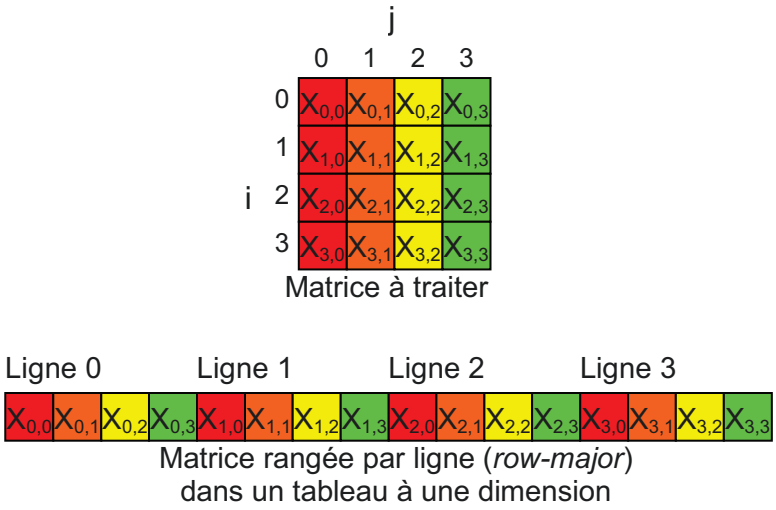



Figure 25 : Système de rangement par ligne d'une matrice dans un tableau à une dimension

Commençons d'abord par un exemple simple pour calculer l'indice d'une donnée d'une matrice stockée dans un tableau. Pour calculer l'indice de la donnée $X_{2,1}$, il suffit de :

- compter le nombre de données par ligne : $n = 4$
- compter le nombre de lignes précédent la ligne contenant la donnée $X_{2,1}$. Ce nombre est le même que l'indice de la ligne contenant la donnée $X_{2,1}$, c'est à dire $i = 2$.
- compter le nombre de colonnes précédent la ligne contenant la donnée $X_{2,1}$. Ce nombre est le même que l'indice de la colonne contenant la donnée $X_{2,1}$, c'est-à-dire $j = 1$.
- calculer le nombre de données précédant la donnée dont on recherche l'indice $k$. Ce nombre de données est le même que l'indice recherché. Par conséquent, pour calculer l'indice, il suffit de calculer le nombre de données (précédant la donnée dont on recherche l'indice) à partir de $n$, $i$ et $j$. On a donc : $k = i * n + j$ . Dans cet exemple, on a donc : $k = 2 * 4 + 1 = 9$ . En effet, en comptant à partir de 0, on remarque que $X_{2,1}$ est bien à l'indice 9 dans le tableau à une dimension (contenant la matrice).

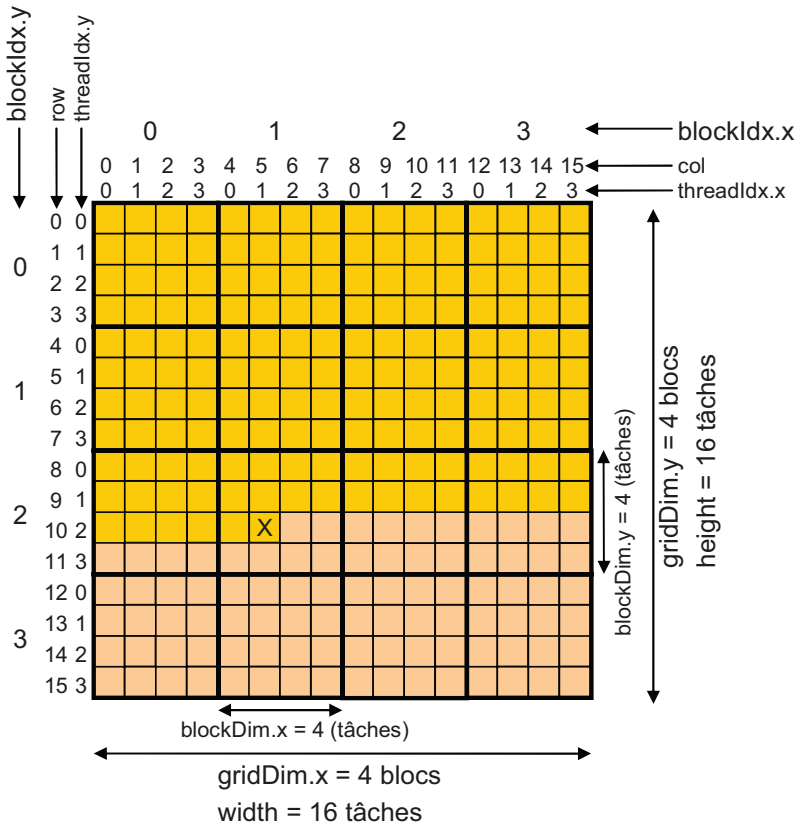
Cette façon de calculer l'indice est généralisable à une matrice résultat si l'on connaît les indices de tâches dans la matrice (indices `row` et `col` décrit précédemment).

Donc pour toute matrice, on obtient l'indice *k* d'une donnée de la matrice (stockée dans un tableau) en remplaçant dans le calcul de *k* précédent :

- *n* par la largeur de la matrice résultat, c'est-à-dire `width`. `width` sera appelée `A.stride` dans le programme.
- *i* par `row`.
- *j* par `col`.

L'indice d'une donnée de la matrice résultat dans un tableau à une dimension est donc : $k = row * width + col$ .

La Figure 25 et la Figure 26 illustrent ce système de calcul d'indice.




Calcul de l'indice de tâche dans une matrice (contenue par un tableau C à une dimension) à partir de ses coordonnées deux dimensions (les lignes de la matrice sont rangées les unes à la suite des autres dans le tableau une dimension) :
k = row * gridDim.x + col = row * width + col
Exemples :
Calcul de l'indice de tableau pour la tâche désignée par « **X** » :
row = blockIdx.y * blockDim.y + threadIdx.y = 2 * 4 + 2 = 10
col = blockIdx.x * blockDim.x + threadIdx.x = 1 * 4 + 1 = 5
k = row * width + col = 10 * 16 + 5 = 165 (numéro de tâche dans un tableau à une dimension commençant par la tâche 0)

Figure 26 : Système de calcul d'indice de la donnée d'une matrice stockée par ligne dans un tableau à une dimension

### II.4.2.4 Opérations exécutées par le kernel au niveau global

L'intérêt de la programmation CUDA est d'exécuter des calculs parallélisables sur carte graphique plus rapidement que sur CPU.

Or on a vu précédemment que l'accès à la mémoire partagée est beaucoup plus rapide que l'accès à la mémoire globale. C'est pourquoi nous privilégierons l'utilisation de la mémoire partagée pour la multiplication de deux matrices.

Cependant, il faut se rappeller que malgré sa rapidité d'accès, la mémoire partagée est très limitée par rapport à la mémoire globale, en particulier :

- la mémoire partagée est localisée physiquement dans chacun des 16 multiprocesseurs d'un processeur graphique. Par conséquent, les données à traiter sont beaucoup plus proches physiquement des cœurs de traitement, ce qui explique sa rapidité d'accès. Mais la mémoire partagée de chaque multiprocesseur est relativement limitée : 48 ko maximum seulement de mémoire partagée accessible à toutes les tâches d'un bloc.
- la mémoire globale possède un espace mémoire très étendu (6 Go), elle est accessible par toutes les tâches d'une grille de tâches. Mais elle est localisée physiquement à l'extérieur du processeur graphique sur la carte graphique, ce qui explique relative lenteur.

C'est pourquoi :

- la mémoire globale aura suffisamment d'espace pour stocker l'intégralité des deux matrices opérandes.
- la mémoire partagée n'aura pas suffisamment d'espace pour stocker l'intégralité des deux matrices opérandes.

Par conséquent, après le transfert des données de l'hôte vers la mémoire globale de la carte graphique, les matrices opérandes devront être découpées en sous-matrices suffisamment petites pour que leurs données puissent être stockées en mémoire partagée.

La Figure 27 et la Figure 28 illustrent un découpage possible du calcul.

Sur ces deux figures, on réalise la multiplication d'une matrice $A$ de taille $3*2$ (blocs) et d'une matrice $B$ de taille $2*3$ (blocs). Le résultat est donc une matrice carrée de taille $3*3$ (blocs).

Si on considère un élément isolé $C_{i,j}$ de la matrice résultat (Fig. 29 et 30), le calcul résulte en $C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j}$

Avec :

- $A_{i,k}$ : élément situé à l'intersection de la ligne $i$ et de la colonne $k$ de la matrice $A$,
- $B_{k,j}$ : élément situé à l'intersection de la ligne $k$ et de la colonne $j$ de la matrice $B$,
- $C_{i,j}$ : élément situé à l'intersection de la ligne $i$ et de la colonne $j$ de la matrice $C$.

Pour ne pas surcharger la mémoire partagée, on réalise le calcul en deux étapes en effectuant la multiplication des sous-matrices :

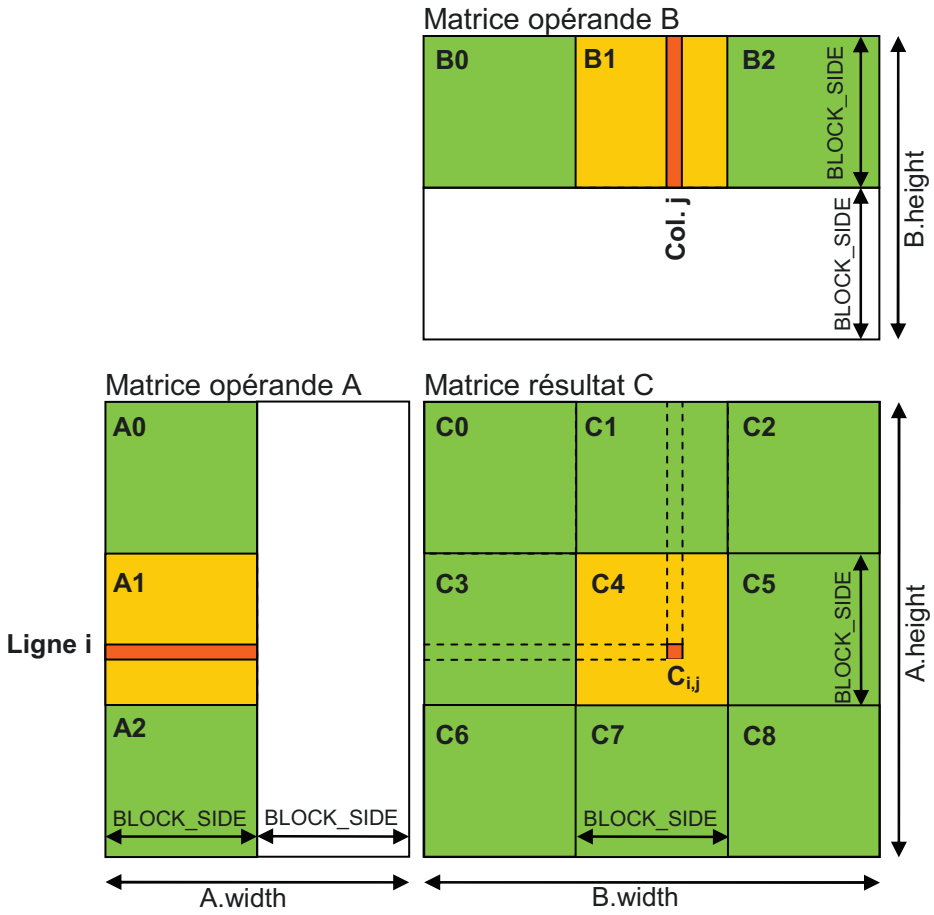
- $A1 * B1$,

puis de :

- $A4 * B4$.

Ces deux étapes au niveau des sous-matrices se traduisent en deux calculs au niveau de l'élément $C_{i,j}$ :

- $[C_{i,j}]_{p1} = \sum_{k=0}^{n/2-1} A_{i,k} B_{k,j}$ pendant la multiplication des sous-matrices $A1 * B1$,
- $[C_{i,j}]_{p2} = \sum_{k=n/2}^{n-1} A_{i,k} B_{k,j}$ pendant la multiplication des sous-matrices $A4 * B4$.




Le bloc de tâches BT4 :

- transfère les blocs opérandes A1 et B1 dans la mémoire partagée du multiprocesseur M4.
- exécute la multiplication de A1 par B1 et stocke le bloc de résultats partiels en mémoire de registres dans le multiprocesseur M4.

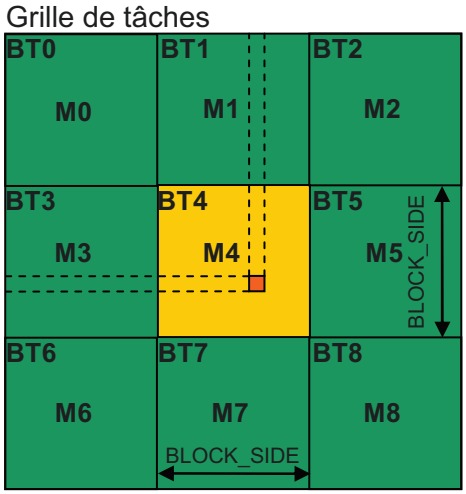
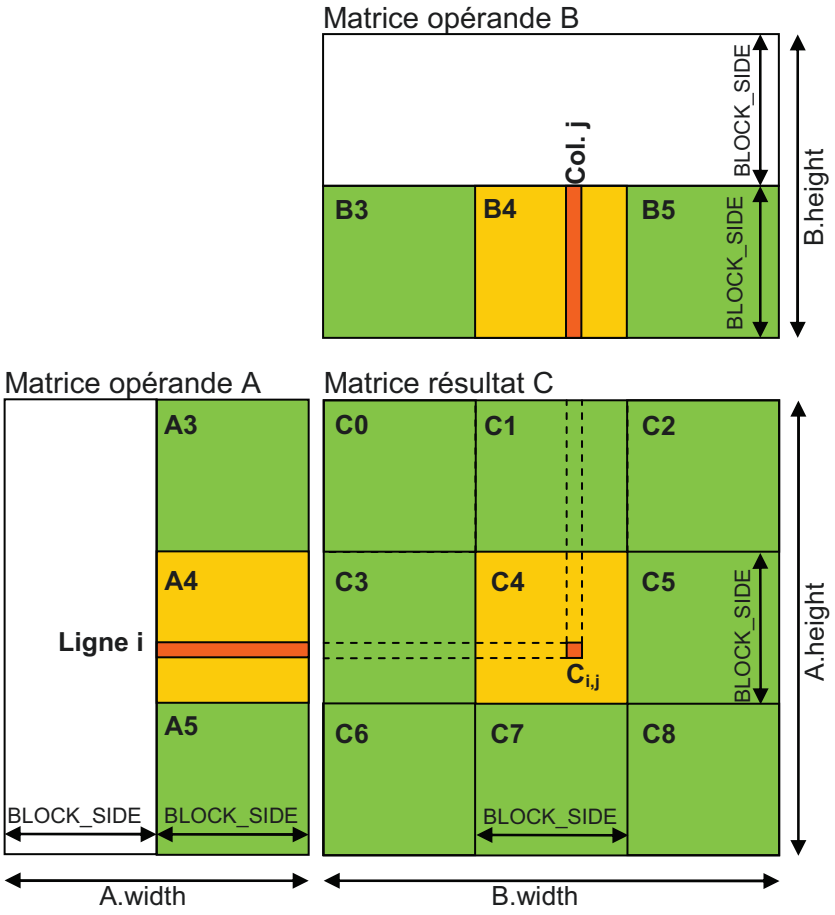



Figure 27 : multiplication de la 1ère colonne de blocs de la matrice A avec la 1ère ligne de blocs de la matrice B.

L'élément $C_{i,j}$ est égal au cumul du résultat des deux étapes.

La répartition des calculs pourra se faire ainsi :

- chargement de $A1$ et $B1$ en mémoire partagée ;
- calcul du résutat partiel de $A1 * B1$ ;
- chargement de $A4$ et $B4$ en mémoire partagée ;
- calcul du résutat partiel de $A4 * B4$ et cumul avec le résultat partiel précédent ;
- transfert des résultats de la mémoire de registre vers la mémoire globale.




Le bloc de tâches BT4 :

- transfère les blocs opérandes A4 et B4 dans la mémoire partagée du multiprocesseur M4.
- exécute la multiplication de A4 par B4 et cumule le bloc de résultats partiels en mémoire de registres dans le multiprocesseur M4.

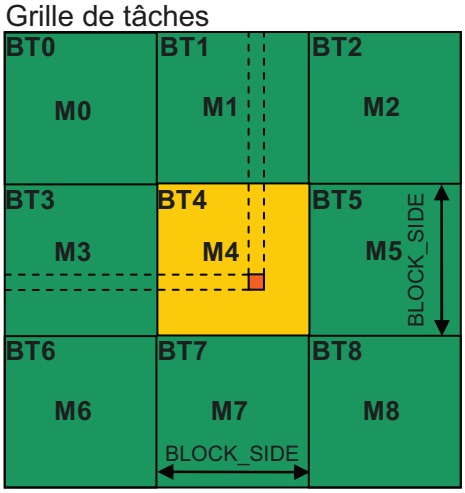



Figure 28 : multiplication de la 2ème colonne de blocs de la matrice A avec la 2ème ligne de blocs de la matrice B.

Dans le cas général, pour réaliser une multiplication de matrices utilisant la mémoire partagée, on devra donc suivre la procédure globale suivante :

- chargement de la 1ère colonne de blocs de la matrice *A* dans les mémoires partagées des multiprocesseurs.
- chargement de la 1ère ligne de blocs de la matrice *B* dans les mémoires partagées des multiprocesseurs.
- multiplication de la 1ère colonne de blocs de la matrice *A* par la 1ère ligne de blocs de la matrice *B*.
- stockage des résultats partiels.
- chargement de la 2ère colonne de blocs de la matrice *A* dans les mémoires partagées des multiprocesseurs.
- chargement de la 2ère ligne de blocs de la matrice *B* dans les mémoires partagées des multiprocesseurs.
- multiplication de la 2ème colonne de blocs de la matrice *A* et de la 2ème ligne de blocs de la matrice *B*.
- cumul des résultats de cette multiplication avec les résultats partiels obtenus précédemment.
- répétition des opérations de transfert, multiplication, cumul jusqu'à arriver au cumul du résultat de la multiplication de la dernière colonne de blocs de *A* par la dernière ligne de blocs de *B*.
- transfert de la matrice résultat vers le tableau prévu pour son stockage en mémoire globale.

#### II.4.2.5 Opérations exécutées par chaque multiprocesseur

La procédure qui vient d'être décrite permettant l'utilisation de la mémoire partagée est une procédure vue d'un point de vue global. La grille de tâches réalise donc l'ensemble des opérations qui vient d'être décrite.

Pour mieux comprendre la manière dont est organisé le calcul, on doit progresser d'un cran dans le niveau de détail des opérations : nous allons donc étudier les opérations exécutées au niveau du multiprocesseur.

Pour simplifier les explications, nous allons considérer le traitement d'un bloc résident par multiprocesseur. On a déjà vu qu'un multiprocesseur peut accepter un ou plusieurs blocs de tâches (selon les ressources disponibles) qu'il traite simultanément : les blocs résidents. Un multiprocesseur peut traiter au maximum 8 blocs résidents.

Dans cet exemple, la grille de tâche est découpée en blocs de tâches par le gestionnaire de tâches. Chaque multiprocesseur reçoit donc un bloc de tâches qui devient bloc résident.

On rappelle que le processeur graphique utilisé pour ce mémoire comporte 14 multiprocesseurs sur les 16 que comporte l'architecture Fermi. Tous les multiprocesseurs disponibles reçoivent donc un bloc de tâches chacun qui devient bloc résident. Une fois qu'un multiprocesseur a exécuté son bloc de tâches résident, le gestionnaire de tâches lui en envoie un suivant qui devient à son tour bloc résident.

Chacun des blocs de tâches est chargé de l'exécution d'une partie des opérations, comme cela a été décrit précédemment (§II.4.2.4). C'est-à-dire qu'un bloc de tâches ne traite :

- qu'une seule ligne de blocs de la matrice *A*,
- qu'une seule colonne de blocs de la matrice *B*.

Chaque bloc de tâches d'un multiprocesseur exécute simultanément les mêmes opérations qu'un autre bloc de tâches d'un autre multiprocesseur mais sur des données différentes.

Chaque bloc de tâches transfère donc une partie des données opérandes dans la mémoire partagée du multiprocesseur dans lequel il se trouve, exécute sa part de calcul, puis transfère une partie de la matrice de résultat de la mémoire de registres du multiprocesseur dans lequel il se trouve vers la mémoire globale de la carte graphique.

Si on se base sur la procédure globale précédente (§II.4.2.4) permettant l'utilisation de la mémoire partagée pour la multiplication de matrice, mais cette fois au niveau d'un bloc isolé, chaque multiprocesseur va traiter son bloc de tâches en suivant plusieurs étapes chronologiques. La Figure 29 illustre ces différentes étapes pour le bloc *BT4* seul (des Figure 27 et Figure 28). Il faut se rappeler que les autres blocs exécutent les mêmes opérations simultanément mais sur des données différentes.

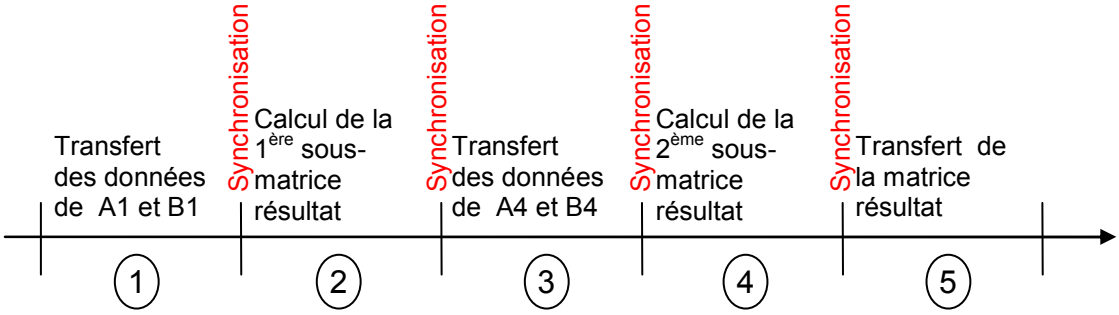



Figure 29 : Chronogramme du calcul du point de vue d'un multiprocesseur

Les 5 étapes de la Figure 29 sont détaillées ci-dessous :

1. Transfert des données de *A1* et *B1* : les tâches d'un bloc prennent en charge le transfert en parallèle des données nécessaires au calcul. Le transfert de l'ensemble des données est réparti dans les différentes tâches.
2. Calcul de la 1ère sous-matrice de résultats partiels : le calcul peut commencer sur chaque tâche à la condition que toutes les tâches sans exception aient terminé le transfert des données. Il est donc nécessaire, à ce moment du calcul, de synchroniser toutes les tâches.
3. Transfert des données de *A4* et *B4*. Le transfert peut commencer à condition que toutes les tâches aient terminé leur calcul. Il est donc à nouveau nécessaire de synchroniser toutes les tâches.
4. Calcul de la 2ème sous-matrice de résultats partiels. Cumul du résultat avec la 1ère sous-matrice de résultats partiels. Ce calcul ne commence que lorsque toutes les tâches ont terminé le transfert des données opérandes de *A4* et *B4*. Une synchronisation est donc à nouveau nécessaire.
5. Transfert du résultat : la sous-matrice résultat est transférée de la mémoire de registres vers la mémoire globale. Ce transfert ne peut commencer que si toutes les tâches ont terminé leur calcul. Une synchronisation est donc à nouveau nécessaire.

### II.4.2.6 Ensemble des opérations exécutées par une tâche isolée

On vient de décrire la procédure montrant l'enchaînement des opérations réalisées par un multiprocesseur isolé pour exécuter son bloc de tâches. Deux blocs étaient multipliés l'un par l'autre.

Pour comprendre encore plus en détails la manière dont est organisé le calcul, on doit encore progresser d'un cran dans le niveau de détail des opérations : nous allons donc étudier les opérations exécutées par une tâche isolée au sein d'un bloc de tâches.

On rappelle d'abord que la mémoire partagée d'un multiprocesseur est accessible à toutes les tâches du bloc de tâches résidant sur ce multiprocesseur. On rappelle que chaque tâche a accès à un certain nombre de registres de la mémoire de registre mais qu'une tâche n'a pas accès aux registres des autres tâches dans la mémoire de registres.

Chacune des tâches d'un bloc de tâches est chargée de l'exécution d'une partie des opérations décrites prédédemment dans la procédure d'excution d'un bloc de tâches. Chaque tâche d'un bloc exécute simultanément les mêmes opérations qu'une autre tâche du même bloc mais sur des données différentes.

Chaque tâche doit :

- assurer le transfert d'une partie des données de la mémoire globale vers la mémoire partagée,
- exécuter sa part de calcul,
- transférer son résultat partiel de son registre de résultat vers la mémoire globale de la carte graphique.

Pour que cela soit possible, on doit déterminer pour chaque tâche quelles données sont transférées, de façon à ce que l'ensemble des transferts réalisés par les tâches effectue bien le transfert de l'ensemble des données.

Chaque tâche doit pouvoir identifier les données à transférer, ce qu'elle peut faire à l'aide des variables prédéfinies `threadIdx`, `blockIdx`, `blockDim` et `gridDim` de façon similaire au stockage des résultats expliqué dans le paragraphe II.4.2.3. La façon la plus rapide de transférer un bloc de données est que les données à transférer soient réparties de façon égale entre les tâches.

Si les données à transférer étaient réparties inégalement, il y aurait un écart de temps beaucoup trop grand entre les transferts pour pouvoir considérer ces transferts comme optimisés. Pour optimiser le transfert, il est donc préférable que la quantité de données à transférer soit un multiple du nombre de tâches par bloc. Ceci permet ainsi une répartition égale des données à transférer entre les tâches.

Dans le cadre de ce mémoire, il a été choisi de transférer autant de données que de tâches, ce qui permet à chaque tâche de calculer l'indice des données à transférer à partir de ces propres indices de tâche. Ce qui veut dire également qu'une tâche isolée ne transfère qu'une seule donnée de la :

- matrice $A$,
- matrice $B$.

Par contre, pour exécuter le produit scalaire d'une ligne de données de la matrice $A$ par une colonne de données de la matrice $B$, une tâche isolée a évidemment besoin non pas d'une seule donnée de chacune des matrice $A$ et $B$ mais :

- d'une ligne de données de la matrice $A$,
- d'une colonne de données de la matrice $B$.

Une tâche isolée a donc besoin de beaucoup plus de données que ce qu'elle transfère, c'est-à-dire qu'une tâche isolée a besoin des données que de nombreuses autres tâches auront transféré.

Cependant, en raison de la structure physique d'une carte graphique, la vitesse de transfert peut varier de quelques fractions de secondes selon les tâches. Certaines tâches auront terminé leur transfert juste avant ou juste après d'autres tâches. Le calcul ne doit débuter que lorsqu'on est sûr que toutes les données ont été transférées, c'est-à-dire que toutes les tâches se déroulant en parallèle dans un bloc ont fini leur transfert. Si les tâches en avance n'attendent pas la fin du transfert des tâches en retard, les tâches en avance ne disposeront pas de toutes les données opérandes nécessaires à leur propre calcul.

Cet écart de de temps de transfert n'étant pas prévisible, on doit demander (à l'aide d'une instruction CUDA prédéfinie) à chacune des tâches d'attendre que toutes les autres tâches aient terminé leur transfert avant de pouvoir autoriser le calcul. On doit donc « synchroniser » les tâches. L'instruction CUDA qui permet de synchroniser les tâches est l'instruction `__syncthreads()` qui est une fonction CUDA prédéfinie. Cette instruction oblige les tâches qui ont fini leurs transferts d'attendre les tâches qui n'ont pas fini le leur.

Le calcul est ensuite effectué. Chaque tâche calcule le résultat partiel d'un élément de la matrice résultat. Le résultat de ce calcul est stocké dans un registre de la mémoire de registres. Ce registre n'est accessible que par cette tâche seule.

La tâche en question va ensuite devoir transférer d'autres données opérandes et calculer le résultat partiel suivant pour le sommer à celui stocké dans la mémoire de registres. Cependant, certaines tâches auront fini leur calcul juste avant d'autres tâches qui seront très légèrement en retard. Si les tâches en avance commencent le transfert des données opérandes suivantes sans attendre que les tâches en retard aient fini leur calcul, il y a le risque que les tâches en retard exécutent leur calcul avec les mauvaises données opérandes, ce qui donnerait un résultat érroné.

Les tâches en avance doivent donc attendre que les tâches en retard aient fini leur calcul avant de commencer le transfert des données opérandes suivantes. Comme précédemment, toutes les tâches doivent donc de nouveau être synchronisées (à l'aide la même instruction CUDA prédéfinie que précédemment) avant de commencer le transfert des données opérandes suivantes.

Pour la 2$^{ème}$ partie du calcul, on retrouve alors un processus similaire aux étapes 1 et 2 de la Figure 29, avec transfert des données, synchronisation et calcul. Le second résultat est accumulé dans le registre contenant le premier résultat partiel.

Dans l'exemple utilisé s'appuyant sur les Figure 27 et Figure 28, le calcul est décomposé en deux calculs partiels (*A1*B1* et *A4*B4*).

Après le cumul du résultat du dernier calcul partiel (qui se trouve être le 2$^{ème}$ calcul partiel dans l'exemple), le résultat stocké dans le registre est alors transféré vers la mémoire globale. Les blocs de données de la matrice résultat étant de la même taille que les blocs de tâches (autant de données que de tâches), il suffit donc d'attribuer une donnée résultat (à transférer) par tâche. L'indice de la donnée résultat dans le tableau stockant la matrice résultat est donc déterminé en fonction des indices identifiant la tâche (`threadIdx.x`, `threadIdx.y`, `blockIdx.x` et `blockIdx.y`) tout comme dans l'étape 1 pour le transfert des données.

Les différentes étapes contribuant à la multiplication de la matrice *A* par la matrice *B* peut être résumée en un seul schéma illustré sur la Figure 30.

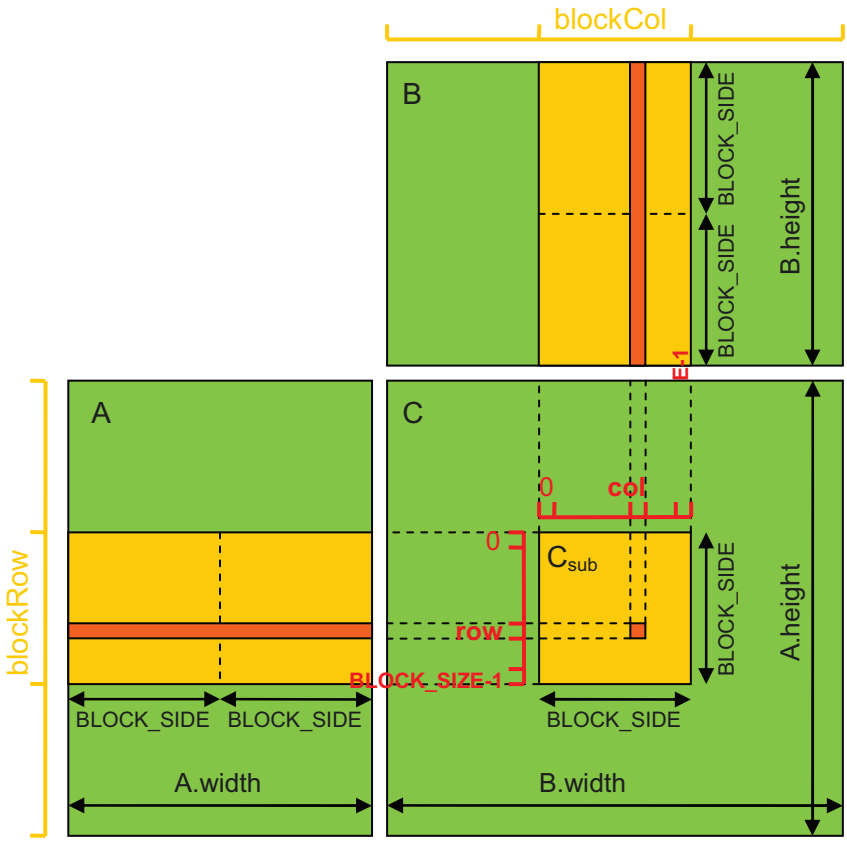



Figure 30 : Multiplication de matrices utilisant la mémoire partagée

### II.4.2.7 Facteurs limitatifs du nombre de blocs résidents

Le nombre de blocs résidents doit satisfaire plusieurs conditions. Dans le cas contraire, le gestionnaire de tâches limitera de lui-même le nombre de blocs résidents jusqu'à ce que les conditions soient satisfaites. Les conditions à satisfaire sont les suivantes :

- à un instant donné, l'ensemble des tâches résidentes ne doit pas nécessiter plus de 32768 registres. C'est l'un des facteurs qui va limiter le nombre de blocs résidents,
- un autre facteur est le nombre de tâches résidentes qui doit toujours être inférieur à 1536 par multiprocesseur,
- la totalité de la mémoire partagée utilisée par les blocs résidents doit toujours rester inférieure à 48 ko par multiprocesseur,
- le nombre de blocs résidents ne peut jamais dépasser 8,
- on ne peut pas attribuer moins de 16 registres par tâche.

Une option du compilateur (***-maxrregcount***) permet de spécifier au compilateur *nvcc* le nombre maximum de registres qu'on souhaite allouer à une tâche. Cependant, comme indiqué ci-dessus, on aura toujours au moins 16 registres utilisés (même si on en spécifie moins de 16) quelle que soit la valeur de ***–maxrregcount***.

La mémoire de registres étant **limitée à 32768 registres par multiprocesseur**, plus il y aura de tâches résidentes, moins il y aura de registres à attribuer par tâche. Le nombre maximal de registres par tâches se déduit donc en divisant le nombre maximal de registres disponibles (par multiprocesseur) par le nombre de tâches résidentes.

Si l'ensemble des tâches résidentes requiert un nombre de registres supérieur aux capacités matérielles d'un multiprocesseur, le gestionnaire de tâches du GPU réduit le nombre de blocs résidents par multiprocesseur jusqu'à ce que la condition sur les registres soit satisfaite. Le nombre de registres à choisir est donc le résultat d'un compromis..

Si la **quantité de mémoire partagée** nécessaire est supérieure à la quantité de mémoire partagée disponible par tâche, cette quantité devient un facteur limitatif du nombre de blocs résidents. C'est-à-dire que le gestionnaire de tâches du GPU va automatiquement réduire le nombre de blocs résidents à attribuer jusqu'à ce qu'ils soient suffisamment peu nombreux pour que la mémoire partagée disponible soit suffisante pour l'ensemble des tâches résidentes [1].

Exemple :

On considère :

- des blocs de 256 tâches,
- 20 registres par tâche,
- chaque tâche utilise 6 nombres flottants, donc 48 octets de mémoire partagée.

La limite des 32768 registres impose d'avoir au plus 6 blocs résidents. En effet, 6 blocs résidents nécessitent :

20 reg./tâche * 256 tâches / bloc * 6 blocs = 30270 registres.

La 2ème condition (moins de 1536 tâches résidentes) impose 6 blocs car :

256 tâches/bloc * 6 blocs = 1536 tâches résidentes

La 3ème condition (moins de 48 ko de mémoire partagée) impose 4 blocs car :

48 octets/tâche * 256 tâches/bloc * 4 blocs = 48 ko

La condition la plus restrictive impose 4 blocs résidents par multiprocesseur.

## II.5 Analyse des résultats pour l'API du moteur d'exécution CUDA

### II.5.1 Présentation des conditions de mesure

Dans le kit de développement CUDA coexistent deux API principales de base pour CUDA™ :

- L'API du moteur d'execution CUDA (*runtime*) dont les noms d'instruction commencent par le terme `cuda` (ex : `cudaMalloc`). Cette API comporte :
  - une interface de bas niveau basé sur le style C
  - une interface de haut niveau basé sur le style C++
- L'API du pilote CUDA dont les noms d'instruction commencent par le terme `cu` (ex : `cuMemAlloc`).

L'API du moteur d'exécution CUDA (*runtime*) met à disposition des fonctions C qui s'exécutent sur l'hôte pour :

- allouer et libérer de la mémoire de périphérique,
- transférer des données entre mémoire hôte et mémoire de périphérique,
- gérer des systèmes comportant plusieurs périphériques.

L'API du moteur d'exécution CUDA (*runtime*) est construit à partir de l'API du pilote CUDA qui est également accessible par programmation. L'API du pilote CUDA offre un niveau de contrôle supplémentaire grâce à certains concepts de niveau inférieur. Ces concepts sont implicites lorsqu'on utilise l'API du moteur d'exécution, le programmeur n'a donc pas besoin de les inclure explicitement dans son programme CUDA-C, ce qui permet d'obtenir un code plus concis.

Typiquement, une application utilise :

- soit l'API du moteur d'exécution CUDA,
- soit l'API du pilote CUDA,

mais elle peut également utiliser les deux API, bien que ce soit moins fréquent.

L'univers CUDA étant très vaste, il a été choisi pour ce projet de n'employer que l'interface de bas niveau de l'API du moteur d'exécution CUDA.

La carte graphique GTX 470 a les capacités de faire le calcul en double précision. Pour ce faire, le compilateur doit être paramétré au moyen de l'option **-arch=sm_20**, ce qui a été fait dans le cadre de cette étude. Le chiffre 20 fait référence à la version 2.0 de l'architecture du processeur graphique.

Dans le cadre de cette étude, les données (opérandes et résultat) sont transmises entre MATLAB et le programme CUDA via une fonction MEX.

Les résultats du calcul de MATLAB et de CUDA ont été analysés et un histogramme de la valeur des erreurs a été établi. Les temps de calcul de CUDA ont également été comparés aux temps de calcul de MATLAB. Le CPU utilisé comprenant 4 cœurs, ce chiffre a été inclus dans une des options MATLAB.

Dans l'environnement de CUDA™, il existe un outil très utile pour analyser les performances d'un programme CUDA : c'est le *profiler* (*Compute Visual Profiler*).

Ce profiler permet d'afficher la plupart des paramètres utilisés par CUDA pour exécuter un programme. Il permet egalement d'afficher des histogrammes (de plusieurs types) à partir de certains paramètres. Dans le profiler, on peut choisir d'afficher certains paramètres et pas d'autres. Les paramètres utilisés pour ce mémoire sont principalement :

- la taille de la grille (en `x`, en `y` et en `z`),
- la taille des blocs (en `x`, en `y` et en `z`),

- les temps d'exécution mesurés sur le CPU ou sur le GPU,
- les temps de transfert mesurés sur le CPU ou sur le GPU,
- le nombre de warps actifs,
- la quantité de mémoire partagée utilisée par bloc,
- le nombre de registres utilisés par tâche,
- le rapport mémoire partagée utilisée sur mémoire partagée disponible,
- le nombre de blocs résidents par multiprocesseur,
- le nombre de tâches résidentes par multiprocesseur,
- le taux d'occupation (nb_warps_résidents / nb_warps_résidents_max). La carte graphique GTX 470 a la capacité de traiter au maximum 48 warps résidents par multiprocesseur. S'il n'y a eu que 16 warps résidents exécutés par multiprocesseur, on en déduit un taux d'occupation de 33%.

La mesure des temps :

- de transfert de données entre CPU et GPU,
- d'exécution du kernel,

a été réalisée à l'aide du profiler. Le programme appelé par le profiler est le programme MATLAB illustré dans l'organigramme de la Figure 31.

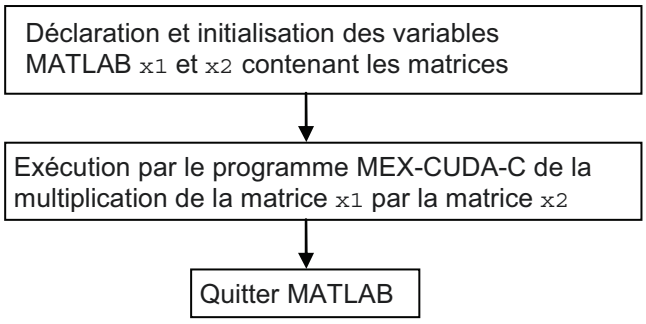



Figure 31 : Programme MATLAB appelé lors de l'utilisation du profiler

Les tests (que ce soit par MATLAB ou par le profiler) mentionnés dans ce mémoire ont tous été réalisés en mode Release. Les essais en mode Debug ne sont pas mentionnés.

Pour pouvoir vérifier les résultats et les erreurs de calcul, il a été établi un programme MATLAB illustré par l'organigramme de la Figure 32.

Pour calculer les temps d'exécution, il a été établi un programme MATLAB illustré par l'organigramme de la Figure 33.

Les temps d'exécution de la fonction MEX-CUDA-C étant relativement fluctuants (écart maximum de l'ordre de 1% de la moyenne des valeurs), les valeurs ont été moyennées sur 10 valeurs.

La première mesure de temps n'a pas été prise en compte car elle inclut les temps de préparation des ressources de calcul (initialisation du pilote et du contexte et chargement de module), donc non représentative du comportement de calcul en régime établi.

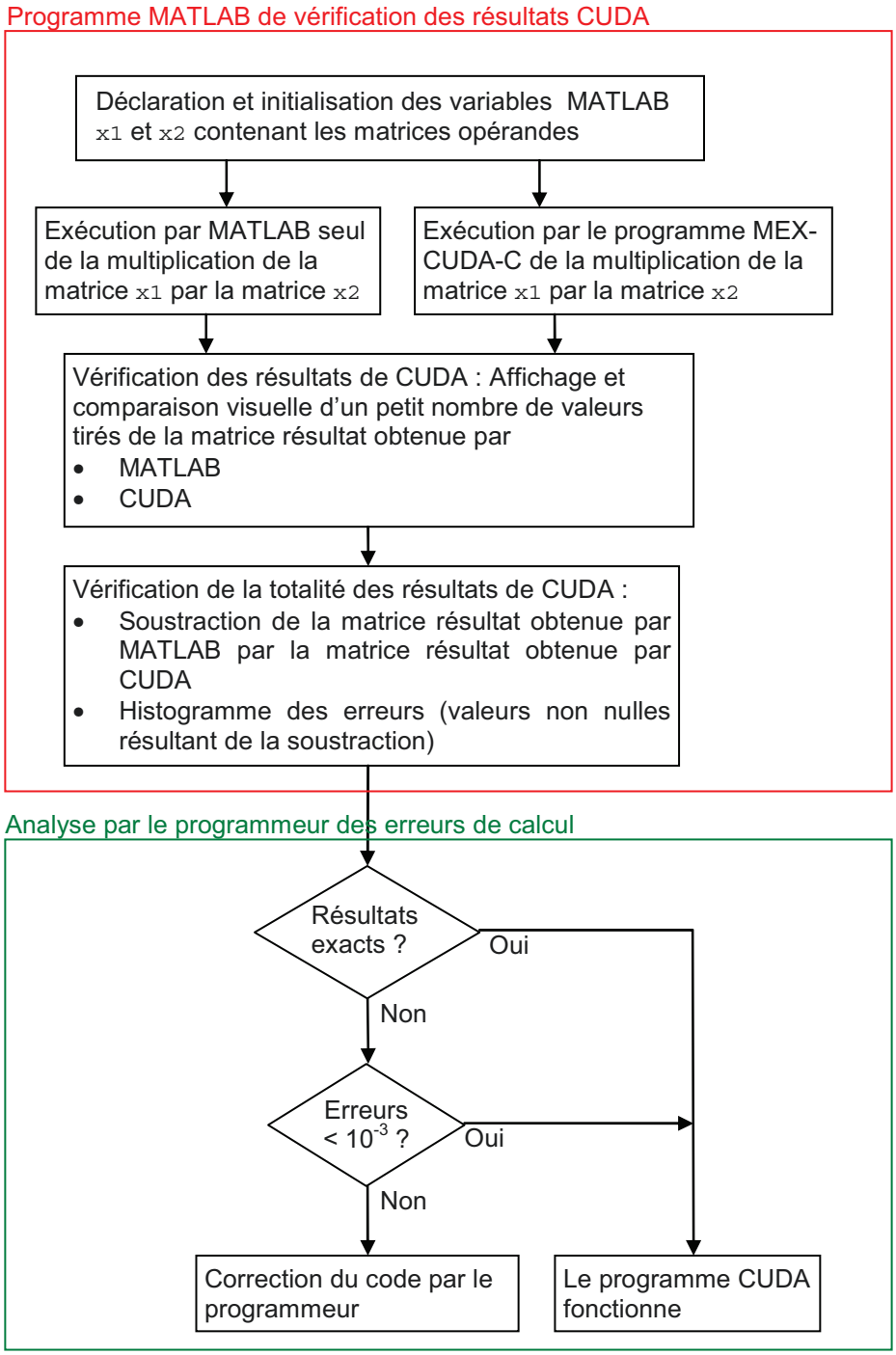



Figure 32 : Vérification de l'exactitude des résultats renvoyés par CUDA

On mesure le temps total d'exécution au moyen de MATLAB, on inclut donc dans la mesure les temps de transfert des données entre MATLAB et la fonction MEX-CUDA-C.

On rappelle que dans un programme MEX-CUDA-C, la taille de la grille de tâches doit être un multiple de la taille des blocs à la fois sur les dimensions `x`, `y` et `z` (voir §I.3.1 et §II.4.2.1). Donc dans le cas d'une grille à deux dimensions, la taille de la matrice résultat doit être un multiple de la taille des blocs à la fois sur les dimensions `x` et `y`.

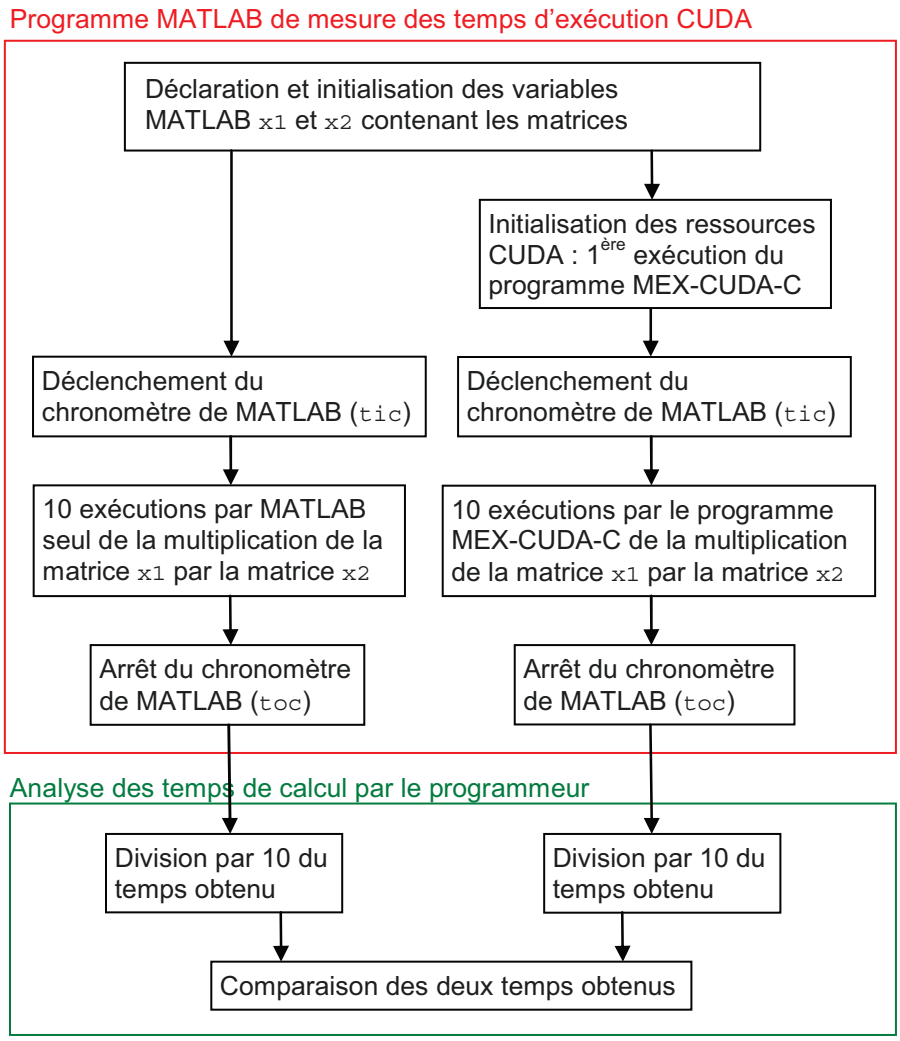



Figure 33 : Mesure des temps d'exécution CUDA et MATLAB

En se basant sur cette seule condition, on pourrait par exemple choisir une grille de taille de $1000\times1000$ avec des blocs de taille $10\times10$. Cependant, il faut se rappeler (voir §I.3.1) que la quantité de tâches par bloc doit être un multiple de la taille de warps, c'est-à-dire multiple de 32. Donner aux blocs une taille non multiple de 32 serait possible puisque des tâches fictives sont ajoutées automatiquement aux warps incomplets mais ce serait contre productif puisque les ressources serait « gaspillées ». De plus le nombre de tâches par bloc ne doit pas dépasser 1024 (données constructeur).

Pour tester le programme pour différentes tailles de blocs tout en satisfaisant les conditions précédentes, le choix est donc assez restreint. Choisir des blocs de tâches ayant le même nombre de tâches en x et en y (bloc carré) n'est pas obligatoire. Mais pour simplifier les explications et réduire la quantité de mesures possibles, il a été choisi des blocs carrés. Il a donc été choisi des blocs carrés de différentes tailles :

- 32 tâches par côté (c'est-à-dire des blocs de 1024 tâches : maximum possible),
- 16 tâches par côté,
- 8 tâches par côté,
- 4 tâches par côté (c'est-à-dire des blocs de 16 tâches entraînant des warps incomplets remplis de tâches fictives).

Par conséquent pour pouvoir observer l'influence de la taille des blocs sur la vitesse de calcul, la taille de la grille doit être multiple à la fois de 32, 16, 8 et 4 (tailles des blocs testés). Les dimensions des matrices testées sont donc $1024 * 1024$, $992 * 992$, $960 * 960$.

Dans les paragraphes qui suivent, nous allons étudier l'influence de divers paramètres sur la performance.

### II.5.2 Influence de la taille de bloc et de la limitation du nombre de registres sur la performance

Il faut ensuite considérer l'influence du nombre de registres par tâche

Les figures 34 à 36 illustrent les temps d'exécution pour 3 tailles de grille de tâche selon deux paramètres :

- le nombre de tâches par côté de bloc,
- le nombre de registres par tâches.

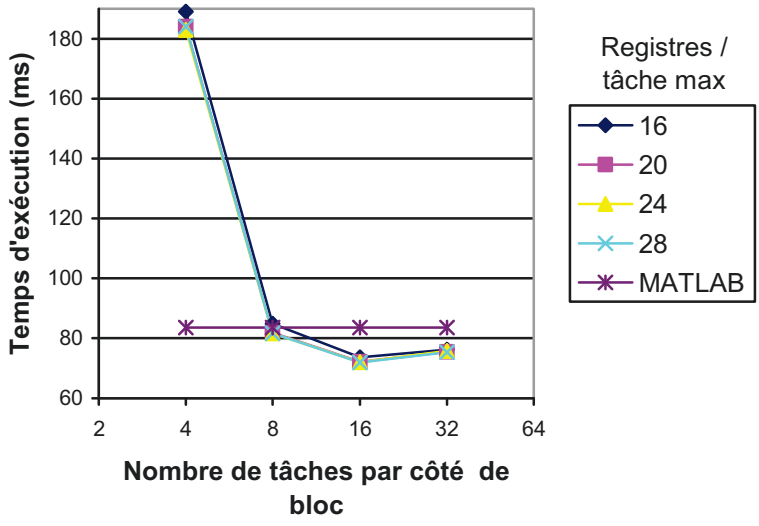



Figure 34 : Temps d'exécution pour une grille de tâches de taille 1024*1024

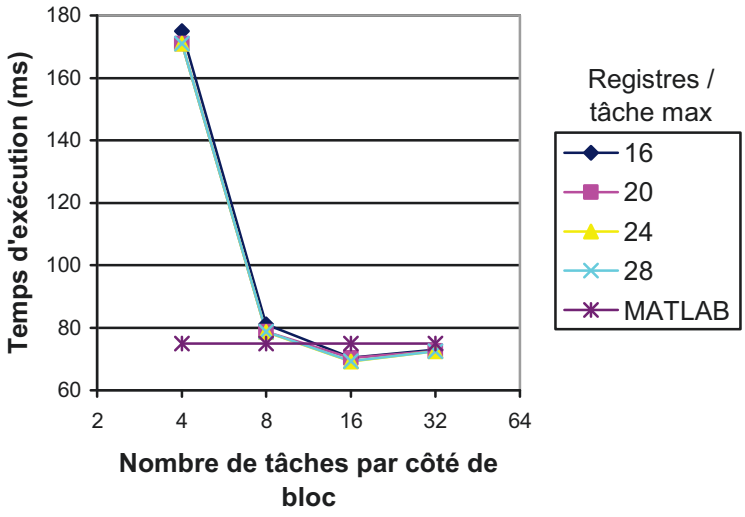



Figure 35 : Temps d'exécution pour une grille de tâches de taille 992*992

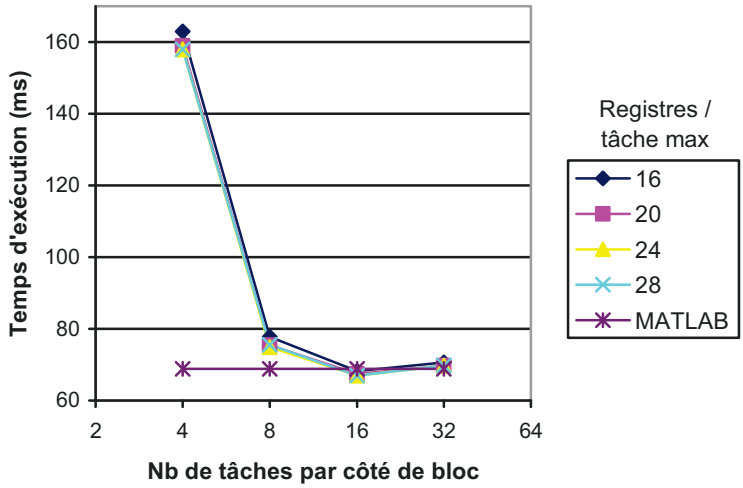



Figure 36 : Temps d'exécution pour une grille de tâches de taille 960*960

D'après les figures 34 à 36, on voit bien que la taille de bloc optimale est $16 \times 16$ puisqu'elle donne les meilleurs temps d'exécution quelle que soit la taille de la grille de tâches.

Discussion sur le nombre de registres :

- on remarque que la limitation du nombre de registres n'a plus d'influence sur la performance au-delà 24 registres,
- cette absence d'influence s'explique à chaque fois que la limitation du nombre de registres est inférieure au nombre de registres qu'une tâche nécessite au maximum. Par exemple, dans notre cas, spécifier une limitation du nombre de registres à un nombre supérieur ou égal à 24 ne change rien car chaque tâche nécessite un nombre maximum de registres inférieur à 24.

On remarque que la courbe pour 16 registres maximum est la seule à se détacher nettement des autres courbes avec des temps de calcul moins bons, et ce quelle que soit la taille de la grille de tâches. Le nombre de registres maximum optimal semble donc être 20, 24 ou 28 car les différences de performance entre 20 registres maximum et 24 registres maximum (et plus) sont si faibles qu'elles peuvent être attribuées aux incertitudes des temps de calcul propres à CUDA.

Tableau 5 : Taux d'occupation des ressources

| Nb de tâches par bloc ↓ | maxrregcount → | 16 | 20 | 24 | 28 |
|---|---|---|---|---|---|
| $4*4$ | Taux d'occupation | 16,7% | 16,7% | 16,7% | 16,7% |
| | Nombre de blocs résidents | 8 | 8 | 8 | 8 |
| $8*8$ | Taux d'occupation | 33,3% | 33,3% | 33,3% | 33,3% |
| | Nombre de blocs résidents | 8 | 8 | 8 | 8 |
| $16*16$ | Taux d'occupation | 100% | 100% | 83,3% | 83,3% |
| | Nombre de blocs résidents | 6 | 6 | 5 | 5 |
| $32*32$ | Taux d'occupation | 66,7% | 66,7% | 66,7% | 66,7% |
| | Nombre de blocs résidents | 1 | 1 | 1 | 1 |

Mais, en observant le Tableau 5, on remarque que pour une taille de blocs de $16*16$, le taux d'occupation est meilleur pour 20 registres maximum que pour 24 registres maximum. Pour les autres tailles de bloc, le nombre de registres n'a pas

d'influence sur le taux d'occupation. Globalement, on peut donc considérer que le nombre de registres maximum optimal est égal à 20.

Finalement, on remarque que le facteur le plus sensible est le choix de la taille des blocs. Le meilleur choix est $16 \times 16$ dans tous les cas. Cependant, l'écart avec MATLAB reste faible.

### II.5.3 Utilisation de la mémoire partagée

Nous avons vu que le nombre de blocs résidents possibles dépend de 5 conditions (voir §II.4.2.7). Nous nous intéressons ici à la condition 3 qui impose de ne jamais dépasser 48 ko pour l'ensemble des blocs résidents d'un multiprocesseur.

Détermination de la mémoire partagée nécessaire pour chaque tâche :

On trouve dans le code du kernel la déclaration des tableaux suivant déclarés en mémoire partagée :

```
__shared__ double As[BLOCK_SIDE][BLOCK_SIDE];
__shared__ double Bs[BLOCK_SIDE][BLOCK_SIDE];
```

Etant donné que le type `double` nécessite 8 octets, la quantité de mémoire par tâche nécessaire pour ces deux tableaux seuls correspond donc à :

$$2 * 8 * BLOCK\_SIDE^2$$

où BLOCK_SIDE correspond à la taille de bloc (taille d'un côté de bloc).

Considérons le pire cas pour lequel BLOCK_SIDE = 32

Dans ce cas, la taille de la mémoire partagée nécessaire est de 16 ko, ce qui reste inférieur à la limite de 48 ko.

Dans notre cas, la limitation de la mémoire partagée à 48 ko n'a pas d'influence sur la détermination du nombre de blocs résidents par le gestionnaire de tâches, puisque quelle que soit la taille de bloc, l'ensemble des tâches ne nécessite jamais plus de 16 ko, ce qui reste très inférieur aux 48 ko disponibles. Il suffit donc de ne prendre en compte que les conditions 1, 2, 4 et 5 (voir §II.4.2.7).

### II.5.4 Influence de la taille de bloc et de la limitation du nombre de registres sur le taux d'occupation des ressources

Les résultats précédents montrent que deux facteurs influent sur les performances mesurées en terme de temps d'exécution :

- principalement le choix de la taille de bloc,
- dans une moindre mesure, le nombre de registres.

On essaie ici d'établir la relation entre ces choix et le taux d'occupation du processeur graphique.

Le taux d'occupation est défini par : $\frac{nbwarpsresidents}{nbwarpsresidents\,\max}$

Dans notre cas, le nombre de warps résidents maximum est de 48. Le nombre de warps résidents se détermine dans chaque cas en tenant compte des cinq conditions à respecter :

- nombre maximum de registres par multiprocesseur,
- nombre maximum de tâches résidentes par multiprocesseur,
- quantité maximum de mémoire partagée par multiprocesseur,
- nombre maximum de blocs résidents par multiprocesseur,

- nombre minimum de registres par tâche.

Ce point a été expliqué au paragraphe II.4.2.7. On a vu que la condition 3 était toujours respectée.

Pour chacun des cas (nombre de blocs, nombre de registres), on peut donc déterminer le nombre de warps résidents choisi par le gestionnaire de tâches, en reprenant la même démarche qu'au paragraphe II.4.2.7.

Pour trouver une relation entre les deux paramètres précédents (taille de bloc et nombre de registres) et le taux d'occupation du processeur graphique, des mesures sont réalisées. On obtient alors les résultats du Tableau 5.

D'après les essais effectués avec le profiler, la taille des grilles de tâche (1024*1024, 992*992, 960*960…) n'a pas d'influence sur les taux d'occupation. Par conséquent, les taux d'occupations mesurés sont regroupés dans le tableau suivant :

Discussion sur le taux d'occupation

Lors des tests sur matrices comportant $16 \times 16$ tâches / bloc et avec nombre de registres limité à 20 par tâche, le taux d'occupation est de 100%. Un taux d'occupation de 100% signifie que pendant le test, le nombre de warps résidents par multiprocesseur est égal au nombre maximum possible, c'est-à-dire 48.

On rappelle que pour exécuter son ou ses blocs de tâches, un multiprocesseur les découpe d'abord en lots de 32 tâches appelés warps. Ces lots de 32 tâches représentent la plus petite découpe possible des blocs de tâches, c'est-à-dire que les tâches sont toujours exécutées par lot de 32.

Dans le cas d'un taux d'occupation de 100%, on a une occupation maximale des ressources, donc 48 warps résidents. Ces 48 warps contiennent au total 1536 tâches, ce qui correspond aux 1536 tâches résidentes maximum ($48 warps * 32 tâches / warp = 1536 tâches$) qu'un multiprocesseur est capable de traiter simultanément.

Il est en général déconseillé d'avoir un taux d'occupation trop bas, cependant un taux d'occupation au maximum (100%) n'est pas forcément synonyme de rapidité étant donné que cette amélioration de taux se fait aux dépens d'autres ressources.

Par exemple, une simple augmentation du nombre maximal de registres de 20 à 24 par tâche (pour des blocs de taille 16*16) se traduit par une diminution du taux d'occupation de 100% à 83,3%. Cette différence est décrite ci-dessous.

Pour mieux comprendre ces résultats, il faut comprendre comment le gestionnaire de tâches du processeur graphique décide du nombre de blocs résidents à attribuer à chaque multiprocesseur.

Considérons des tailles de bloc de 16*16 (256 tâches) :

Quelque soit le nombre de registres nécessaires, la condition sur le nombre de tâches (voir §II.4.2.7) n'est pas respectée pour des blocs de taille 16*16 puisque 8 blocs de 256 tâches dépassent la limite de 1536 tâches maximum. Le gestionnaire diminue donc le nombre de blocs résidents jusqu'à 6 blocs résidents.

Dans le cas où on a 20 registres par tâches pour 6 blocs résidents, 30720 registres sont nécessaires au total, ce qui respecte la condition sur le nombre maximum de registres. Les autres sont respectées. Les 6 blocs résidents de 256 tâches (c'est-à-dire 1536 tâches ou 48 warps) entraîneront donc un taux d'occupation de 100%.

Dans le cas où on a 24 registres par tâches pour 6 blocs résidents, 36864 registres sont nécessaires au total, ce qui ne respecte pas la condition sur le nombre de registres. Le gestionnaire de tâches diminuera donc jusqu'à 5 le nombre de blocs résidents pour que cette condition soit respectée. Ces 5 blocs résidents (constituant un total de 1280 tâches ou 40 warps) entraîneront un taux d'occupation de 83,3%.

Pour des blocs de 256 tâches, on peut donc résumer les calculs précédents par :

- 20 registres par tâche entraînent un taux d'occupation de 100%.
- 24 registres par tâche entraînent un taux d'occupation de 83,3%.

On a donc démontré qu'une augmentation de 4 du nombre de registres par tâche n'améliore pas forcément les performances puisque dans ce cas précis, cette augmentation entraîne une diminution du taux d'occupation. Un taux d'occupation inférieur à 100% signifie que les ressources matérielles d'un multiprocesseur ne sont pas exploitées à leur maximum.

Cas particulier des blocs de petite taille

Pour les cas étudiés jusqu'à maintenant, c'est-à-dire pour des tailles de bloc supérieures à 32, on déduisait le nombre de warps résidents à partir du nombre de tâches résidentes. Cette déduction n'est plus valable pour des tailles de bloc très petites (Exemple : blocs de $4 * 4$ tâches) car un warp ne peut pas englober les tâches de plusieurs blocs.

Un warp ne peut pas contenir moins de 32 tâches. Donc, comme on l'a vu précédemment (voir §I.3.1), si un bloc contient moins de 32 tâches, des tâches fictives sont ajoutées par le multiprocesseur pour constituer un warp de 32 tâches. Dans ce cas précis, on aura donc des warps de 32 tâches ne comportant que 16 tâches réellement utiles au calcul.

Le taux d'occupation se calcule toujours à partir du nombre de warps résidents. On a ici 8 blocs résidents de 16 tâches chacun. On aura donc 8 warps résidents par multiprocesseur, chaque warp contenant la totalité des tâches d'un bloc. Le calcul du taux d'occupation donne donc : *nbwarpsrésidents / nbwarpsrésidentsmax = 8 / 48 = 0,166* c'est-à-dire 16,6%.

C'est pourquoi, pour éviter de gaspiller des ressources de calculs avec des warps incomplets, il est conseillé de donner des tailles de bloc multiples du nombre de tâches par warp, donc multiples de 32, et éviter les blocs contenant moins de 32 tâches.

### II.5.5 Performances ramenées en GFLOPS

Pour pouvoir comparer les performances du processeur graphique lors du traitement de grille de tâches de grandeurs différentes, on convertit en GFLOPS les temps obtenus. Le calcul d'un élément de base d'une matrice résultat nécessite $n$ multiplications et $(n-1)$ additions, $n$ étant le nombre de :

- colonnes de la 1ère matrice opérande
- lignes de la 2ème matrice opérande

Ce qui donne $2*n - 1$ opérations pour calculer un élément de la matrice résultat. Dans le cas où la matrice résultat a également pour dimensions $n \times n$, le nombre d'opérations réalisées au total est : $n * n * (2 * n - 1) = 2 * n^3 - n^2$

Comme $n$ est grand, on peut négliger le terme $n^2$ devant $2*n^3$ (erreur relative $1/2n \approx 5 * n^{-4}$)

Donc on obtient $NBFLOPS = \frac{2 * n^3}{tpscalcul}$

Les tableaux 6 à 9 présentent les résultats obtenus pour respectivement une grille de taille $1024 * 1024$, $992 * 992$, $960 * 960$, $896 * 896$ (multiple de 14).

Tableau 6 : Performances de CUDA (en GFLOPS) pour une grille de taille 1024*1024

| Registres par tâche (max) / Taille de bloc | 16 | 20 | 24 | 28 |
|---|---|---|---|---|
| $4 * 4$ | 11,36 | 11,67 | 11,73 | 11,67 |
| $8 * 8$ | 25,29 | 26,22 | 26,32 | 26,32 |
| $16 * 16$ | 29,14 | 29,74 | 29,78 | 29,87 |
| $32 * 32$ | 28,18 | 28,44 | 28,33 | 28,48 |
| MATLAB seul | 25,69 | | | |

Tableau 7 : Performances de CUDA (en GFLOPS) pour une grille de taille 992*992

| Registres par tâche (max) / Taille de bloc | 16 | 20 | 24 | 28 |
|---|---|---|---|---|
| $4 * 4$ | 11,16 | 11,42 | 11,42 | 11,42 |
| $8 * 8$ | 24,07 | 24,81 | 24,84 | 24,81 |
| $16 * 16$ | 27,73 | 27,77 | 28,21 | 28,17 |
| $32 * 32$ | 26,74 | 26,89 | 26,93 | 26,93 |
| MATLAB seul | 26,07 | | | |

Tableau 8 : Performances de CUDA (en GFLOPS) pour une grille de taille 960*960

| Registres par tâche (max) / Taille de bloc | 16 | 20 | 24 | 28 |
|---|---|---|---|---|
| $4 * 4$ | 10,86 | 11,13 | 11,2 | 11,2 |
| $8 * 8$ | 22,74 | 23,44 | 23,59 | 23,41 |
| $16 * 16$ | 25,91 | 26,21 | 26,41 | 26,41 |
| $32 * 32$ | 25,03 | 25,35 | 25,35 | 25,35 |
| MATLAB seul | 25,68 | | | |

Tableau 9 : Performance de CUDA (en GFLOPS) pour une taille de grille multiple du nombre de multiprocesseurs (896*896)

| Registres par tâche (max) / Taille de bloc | 16 | 20 | 24 | 28 |
|---|---|---|---|---|
| $4 * 4$ | 10,28 | 10,5 | 10,5 | 10,5 |
| $8 * 8$ | 20,32 | 20,88 | 20,88 | 20,85 |
| $16 * 16$ | 22,85 | 23,02 | 23,13 | 23,09 |
| $32 * 32$ | 22,3 | 22,37 | 22,41 | 22,2 |
| MATLAB seul | 25,6 | | | |

Si on compare les Tableau 6, Tableau 7, Tableau 8 et Tableau 9 avec le Tableau 5, on constate que les meilleures performances sont obtenues pour des tailles de bloc de 16*16, c'est-à-dire des tailles de blocs correspondant aux meilleurs taux d'occupation, bien que le nombre de registres ait également une légère influence sur les taux d'occupation.

Si on veut optimiser le calcul, il est donc intéressant de faire l'analyse du nombre de blocs résidents en fonction des choix (taille de bloc, nb de registres, quantité de mémoire par tâche).

En comparant le Tableau 9 avec les Tableau 6, Tableau 7 et Tableau 8, on constate que le fait d'avoir une taille de grille multiple de 14 n'augmente pas la performance en GFLOPS. De plus, pour une taille de grille de taille $896 * 896$, MATLAB exécute plus de GFLOPS que CUDA. Donc, il n'y a finalement aucun intérêt à choisir une taille de grille multiple du nombre de multiprocesseurs.

Sur les 4 tableaux, on peut constater que la performance dépend avant tout de la taille de la grille. On voit bien qu'une grande quantité de tâches par grille est un facteur bien plus déterminant pour la performance qu'une grille multiple du nombre de multiprocesseurs. Plus la grille est grande plus le processeur est rapide.

### II.5.6 Performances de la fonction MEX-CUDA-C comparée aux performances de MATLAB seul

A partir des mesures de performance précédentes, on remarque que CUDA obtient des performances optimales lorsque :

- le nombre de registre est limité à 20.
- les blocs sont limités à une taille de $16 * 16$.

Maintenant qu'on connait les paramètres optimaux, il peut être intéressant de comparer les performances la fonction MEX-CUDA-C et de MATLAB utilisé seul. Pour une taille de bloc de $16 * 16$ et un nombre de registres limité à 20, on obtient les résultats suivants:

Tableau 10 : Comparaison de la performance pour différentes tailles de matrice résultat

| Taille de la matrice résultat | Fonction MEX-CUDA-C | | MATLAB utilisé seul | |
|---|---|---|---|---|
| | Temps d'exécution (ms) | GFLOPS | Temps d'exécution (ms) | GFLOPS |
| $1024 * 1024$ | 71,7 | **29,95** | 83,6 | 25,69 |
| $992 * 992$ | 69,4 | **28,13** | 75,1 | 26,00 |
| $960 * 960$ | 66,8 | **26,49** | 69,0 | 25,64 |
| $928 * 928$ | 64,3 | 24,86 | 62,7 | **25,49** |
| $896 * 896$ | 62,0 | 23,20 | 56,0 | **25,69** |
| $864 * 864$ | 60,1 | 21,46 | 50,7 | **25,44** |
| $832 * 832$ | 58,1 | 19,83 | 46,3 | **24,88** |
| $800 * 800$ | 56,2 | 18,22 | 41,1 | **24,91** |
| $768 * 768$ | 54,4 | 16,65 | 36,1 | **25,10** |
| $736 * 736$ | 52,5 | 15,19 | 31,9 | **25,00** |
| $704 * 704$ | 51,0 | 13,68 | 28,6 | **24,40** |

Dans le Tableau 10, le meilleur des deux cas est mis en gras pour chaque taille de matrice résultat.

On constate donc que la fonction MEX-CUDA-C a une performance supérieure à MATLAB uniquement pour les matrices résultat de taille supérieure ou égale à $960 * 960$.

On constate également que le nombre de GFLOPS exécutés par MATLAB est relativement stable sur l'ensemble des tailles de matrice résultat étudiées alors que le nombre de GFLOPS exécutés par la fonction MEX-CUDA-C diminue régulièrement à mesure que les tailles de matrice résultat diminuent.

### II.5.7 Synthèse des résultats obtenus

Dans le cas d'un problème de multiplication de matrices tel que traité dans ce mémoire, à partir des différentes mesures précédentes, on déduit que pour avoir des résultats optimaux, on doit :

- régler à 20 le nombre maximal de registres utilisables par tâche,
- régler à $16 * 16$ la taille des blocs de tâches,
- utiliser CUDA uniquement dans les cas ou la matrice résultat est de taille supérieure ou égale à $960 * 960$ .

Dans les autres cas de figure (matrices opérandes de taille différente de la taille de la matrice résultat, matrices résultat non carrées, etc.), deux démarches principales sont possibles :

- faire des tests avec le profiler en observant la manière dont évoluent les résultats et les paramètres,
- déduire le nombre de blocs résidents (par multiprocesseur) et de tâches résidentes choisis par le gestionnaire de tâche en sachant que :
  - le gestionnaire de tâches choisit toujours un nombre de blocs résidents le plus élevé possible sans jamais dépasser 8,
  - le gestionnaire de tâches vérifie toujours que les ressources requises par plusieurs blocs de tâches sont disponibles dans les multiprocesseurs ciblés. C'est-à-dire que l'ensemble des futurs blocs résidents d'un multiprocesseur ne doit pas nécessiter plus de :
    - 48 ko de mémoire partagée,
    - 32768 registres,
    - 1536 tâches.

    Le gestionnaire de tâches détermine automatiquement le nombre de blocs résidents en fonction de ces conditions.

Dans le Tableau 11, on peut observer un récapitulatif des temps d'exécution et de transfert de la fonction MEX-CUDA-C. On remarque que la proportion du temps de transfert augmente (faiblement) linéairement avec la taille de la matrice résultat.

Tableau 11 : Synhtèse des résultats pour des blocs de taille 16*16

| Taille de la matrice résultat | Fonction MEX-CUDA-C | | | | | |
|---|---|---|---|---|---|---|
| | Temps d'exécution (ms) | Kernel seul (ms) | Temps de transfert CPU → GPU (ms) | Temps de transfert GPU → CPU (ms) | Temps total (kernel + transferts) (ms) | Proportion temps transfert / temps total (%) |
| 1024*1024 | 71,7 | 21,7 | 3,64+3,53 | 4,46 | 33,3 | 16,2 |
| 960*960 | 66,8 | 17,9 | 3,34+3,15 | 4,02 | 28,4 | 15,7 |
| 896*896 | 62,0 | 14,5 | 2,90+2,80 | 3,50 | 23,7 | 14,8 |

# III Utilisation de la bibliothèque CUBLAS

## III.1 Présentation de la bibliothèque CUBLAS

La bibliothèque CUBLAS [20] est une mise en œuvre de l'ensemble des fonctions standardisées BLAS [21] (*Basic Linear Algebra Subprograms*) au dessus du moteur d'exécution (*runtime*) NVIDIA®CUDA™.

L'ensemble des fonctions BLAS (Basic Linear Algebra Subprograms) est un ensemble de fonctions standardisées réalisant des opérations de base de l'algèbre linéaire, comme des multiplications de vecteurs ou de matrices. Largement utilisées pour le calcul haute performance, ces fonctions ont été développées de manière très optimisées par des constructeurs tels que Intel.

Les fonctions de la bibliothèque BLAS sont réparties en 3 niveaux : 1, 2 et 3.

- Le niveau 1 contient :
  - les opérations sur les vecteurs de la forme $y = \alpha * x + y$ où $\alpha$ est un scalaire (constante) et $x$ et $y$ sont des vecteurs,
  - les opérations produit scalaire et norme, parmi tant d'autres.
- Le niveau 2 contient entre autres :
  - les opérations de type *matrice-vecteur* de la forme $y = \alpha * A * x + \beta * y$ où $\alpha$ et $\beta$ sont des scalaires (constantes), $x$ et $y$ sont des vecteurs,
  - la résolution de $y = T * x$, où $T$ est une matrice triangulaire.
- Le niveau 3 contient entre autres :
  - les opérations de type *matrice-matrice* de la forme $C = \alpha * A * B + \beta * C$ où $\alpha$ et $\beta$ sont des scalaires (constantes), et $A$, $B$ et $C$ sont des matrice,
  - la résolution de $B = \alpha * T^{-1} * B$, où $T$ est une matrice triangulaire.

    Le niveau 3 contient notamment l'opération de multiplication de matrices générales (DGEMM).

La bibliothèque CUBLAS permet à l'utilisateur d'accéder aux ressources de calcul du processeur graphique. Cette bibliothèque permet donc d'exécuter des fonctions de la bibliothèque BLAS dans l'environnement CUDA du processeur graphique.

Pour utiliser cette bibliothèque, l'application doit :

- allouer de l'espace mémoire GPU pour les matrices ou vecteurs requis,
- remplir cet espace mémoire de données,
- appeler la séquence de fonctions CUBLAS souhaitées,
- puis renvoyer les résultats de l'espace mémoire GPU vers l'hôte.

Cette bibliothèque fournit également un ensemble de fonctions prédéfinies (*Helper Function*) destinées à l'écriture et la lecture de données sur le processeur graphique. Cet ensemble de fonctions permet en particulier de gérer :

- les espaces mémoire,
- l'utilisation de la bibliothèque,
- les déplacements de données.

La bibliothèque CUBLAS fournit également un ensemble de fonctions réparties en 3 niveaux calqués sur les 3 niveaux de la bibliothèque de fonctions BLAS précédemment décrits. A l'intérieur des niveaux de l'API CUBLAS, chacune des fonctions est classée selon que les données manipulées sont :

- réelles ou complexes,

- simple ou double précision.

Cette répartition en niveaux et en types de données se base sur la répartition en niveaux et en types de données de la bibliothèque de fonctions standardisées BLAS précédemment décrite. Les niveaux de l'API CUBLAS sont les suivants :

- **BLAS1** : une liste de fonctions de calcul exécutant des opérations sur nombre scalaire ou sur vecteurs. Les opérations sur les vecteurs sont de la forme : $y = \alpha * x + y$ ($x$ et $y$ étant des vecteurs et $\alpha$ une constante).
- **BLAS2** : une liste de fonctions de calcul exécutant des opérations *matrice-vecteur* de la forme $y = \alpha * A * x + \beta * y$ ou de la forme $y = T * x$ ($A$ étant une matrice, $x$ et $y$ des vecteurs et $\alpha$ et $\beta$ des constantes).
- **BLAS3** : une liste de fonctions de calcul exécutant des opérations *matrice-matrice* de la forme $C = \alpha * A * B + \beta * C$, ou de la forme $B = \alpha * A^T * B$ ($A$ et $B$ étant des matrices, et $\alpha$ et $\beta$ des constantes).

Les principales parties d'un programme CUBLAS de base sont similaires à celle d'un programme CUDA, c'est à dire :

- déclaration et initialisation des pointeurs CPU,
- déclaration des pointeurs GPU,
- réservation de mémoire pour les pointeurs GPU,
- transfert des données CPU vers GPU,
- calculs sur GPU,
- transfert des résultats GPU vers CPU,
- libération de la mémoire GPU,
- libération de la mémoire CPU.

La principale différence est que dans un programme CUDA, le calcul est exécuté par un kernel découpé en une grille de tâches et de blocs parallèles définis par le programmeur. Dans un programme CUBLAS, le programmeur ne peut plus définir la taille de grille ou la taille de blocs de tâches, c'est le processeur graphique qui s'en charge automatiquement.

Une seule fonction CUBLAS prédéfinie exécute non plus un nombre réduit d'opérations comme pour une fonction CUDA prédéfinie mais tout un ensemble de calculs sur des matrices, des vecteurs ou des nombres scalaires.

Pour CUBLAS, le paramétrage du calcul se limite donc au passage d'arguments devenus beaucoup plus nombreux que pour les fonctions CUDA. Par exemple la fonction `cublasDgemm()` (correspondant à la fonction DGEMM de la bibliothèque BLAS) nécessite le passage de 13 arguments :

```
void cublasDgemm (char transa, char transb, int m, int n, int k, double
alpha, const double *A, int lda, const double *B, int ldb, double beta,
double *C, int ldc)
```

## III.2 Description d'un programme CUBLAS de base

Dans ce programme de base, nous étudierons l'addition de deux matrices. Ce programme est décrit dans les lignes ci-dessous.

Pour pouvoir additionner deux matrices, elles doivent avoir les mêmes dimensions. On définit donc d'abord le nombre de lignes et de colonnes des matrices à traiter avec les instructions :

```
#define M 5 /* Nombre de lignes */
#define N 4 /* Nombre de colonnes */
```

### III.2.1 Déclaration des pointeurs CPU et GPU

Au début de la fonction principale `main`, on déclare les variables et les pointeurs CPU et GPU :

```
/* Déclaration des pointeurs */
double *x,*y,*z; /* CPU */
double *dx,*dy; /* GPU */
```

Puis on alloue de la mémoire sur l'hôte pour les trois pointeurs CPU :

```
/* Allocation de mémoire sur l'hôte pour les pointeurs de la première et deuxième matrice opérande et pour le pointeur de la matrice résultat */
x = (double*)malloc(M*N*sizeof(double));
y = (double*)malloc(M*N*sizeof(double));
z = (double*)malloc(M*N*sizeof(double));
```

### III.2.2 Initialisation des pointeurs CPU

Dans l'exemple traité, on remplit arbitrairement :

- la première matrice avec les valeurs `0`, `1`, …`M*N`.
- la deuxième matrice avec les valeurs `2`*(`0`, `1`, …`M*N`).

On initialise donc les données pointées par `x` et `y` de la manière suivante :

```
/* Initialisation de x et y */
for(j = 0; j < N; j++)
 {
 for(i = 0; i < M; i++)
        {
        x[IDX2C(i,j,M)] = j*M + i;
        y[IDX2C(i,j,M)] = 2*(j*M + i);
        }
 }
```

Cette boucle permet d'initialiser le tableau :

- `x` par la suite de chiffre allant de `0` à `M*N`,
- `y` par deux fois la suite de chiffres allant de `0` à `M*N`.

`x` et `y` étant en réalité deux vecteurs, on doit calculer la valeur de l'indice de vecteur à partir des deux indices (ligne `i`, colonne `j`) de chaque matrice. On utilise pour cela la macro `IDX2C(i,j,M)` définie ainsi :

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

Cette macro permet de calculer l'indice de la donnée dans un tableau à une dimension à partir des indices ligne et colonne.

Dans cette macro, `ld` représente le nombre de lignes de la matrice [20].

Contrairement à CUDA où les données d'une matrice sont stockées ligne par ligne dans un tableau ; avec CUBLAS, les données d'une matrice sont stockées colonne par colonne dans un tableau. La Figure 37 illustre la différence entre CUDA et CUBLAS

en ce qui concerne la manière de stocker les données d'une matrice dans un tableau à une dimension.

Pour CUBLAS, quand on « lit » dans l'ordre les valeurs du tableau (contenant la matrice), on « lit » d'abord les éléments de la première colonne (colonne 0 du haut vers le bas) de la matrice, puis les éléments de la deuxième colonne, et ainsi de suite jusqu'à la $N^{ième}$ colonne (colonne N-1).

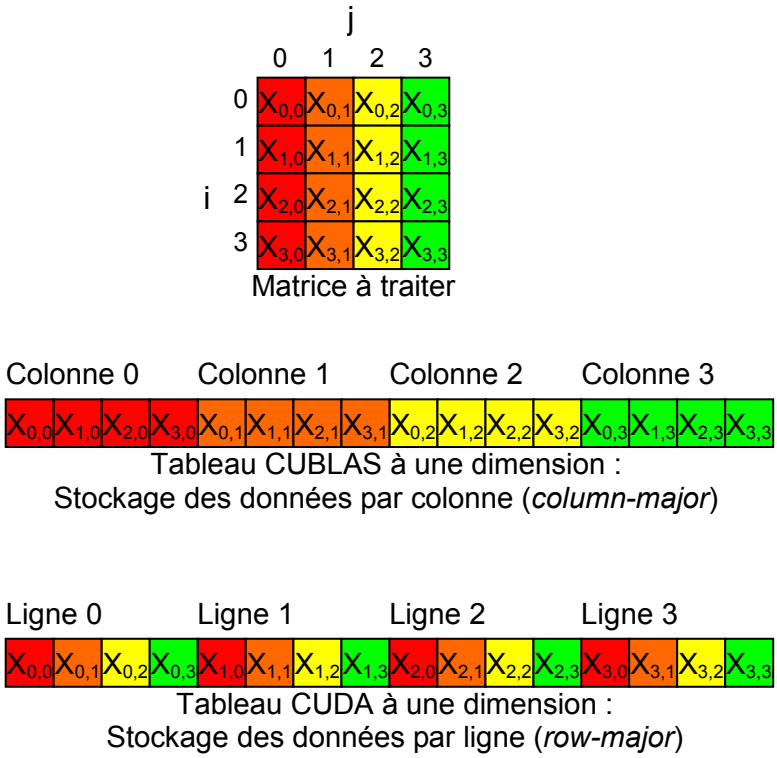



Figure 37 : Rangement d'une matrice par ligne ou par colonne dans un vecteur

Une fois les vecteurs initialisés et avant d'utiliser les fonctions CUBLAS, on doit initialiser la bibliothèque grâce à la fonction :

```
cublasInit();
```

### III.2.3 Réservation de mémoire pour les pointeurs GPU

Une fois la bibliothèque initialisée, on réserve de l'espace mémoire sur la carte graphique avec la fonction `cublasAlloc()` :

```
cublasAlloc(M*N, sizeof(double), (void**)&dx);
cublasAlloc(M*N, sizeof(double), (void**)&dy);
```

Cette fonction crée en espace mémoire GPU un objet capable de contenir un tableau de `M*N` éléments, dans lequel chaque élément nécessite `sizeof(double)` octets de stockage. Si l'appel de fonction ne renvoie pas d'erreur, un pointeur sur l'objet dans l'espace mémoire est placé dans `&dx`.

A noter que ce pointeur est prédéfini pour pointer sur un périphérique de traitement (carte graphique). La fonction `cublasAlloc()` étant basée sur la fonction `cudaMalloc()`, les pointeurs de périphérique de traitement renvoyés par `cublasAlloc()` peuvent par conséquent être passés à n'importe quel kernel de périphérique de traitement CUDA, et non pas seulement à des fonctions CUBLAS.

### III.2.4 Transfert des données CPU vers GPU

Une fois la mémoire allouée sur la carte graphique, on peut transférer les données d'entrée sur la carte avec les instructions :

```
cublasSetVector(M*N, sizeof(double), x, 1, dx, 1);
cublasSetVector(M*N, sizeof(double), y, 1, dy, 1);
```

La fonction `cublasSetVector()` transfère :

- le contenu (1ère matrice opérande) du tableau `x` (CPU) vers le tableau `dx` (GPU), c'est-à-dire $M * N$ éléments,
- le contenu (2ème matrice opérande) du tableau `y` (CPU) vers le tableau `dy` (GPU), c'est à dire $M * N$ éléments.

L'intervalle de stockage (appelé incrément) entre éléments consécutifs est :

- `1` (4ème argument) pour le vecteur source `x`,
- `1` (6ème argument) pour le vecteur destination `dx`.

Cet intervalle de stockage permet de dire de combien on doit incrémenter le pointeur entre deux lectures ou entre deux écritures.

Par exemple, si on avait pris un intervalle de stockage de 4 pour la lecture du tableau x, la 1ère du tableau aurait été lue, puis la 5ème valeur aurait été lue, puis la 9ème, la 13ème, etc. Cet intervalle de stockage fonctionne de manière identique pour le tableau `dx` situé sur la carte graphique.

Donc étant donné que les matrices sont stockées colonne par colonne dans les tableaux, un intervalle de stockage :

- de 1 permet d'accéder aux éléments d'une colonne particulière de la matrice,
- égal au nombre de lignes de la matrice permet d'accéder à une ligne particulière d'une matrice.

Par exemple, en se basant sur la Figure 37, si on donne un intervalle de stockage de 4 à une matrice comportant 4 lignes et 4 colonnes, le pointeur de donnée accèdera à la i$^{ème}$ valeur, puis à la $(i+4)^{ème}$ valeur, puis à la $(i+4+4)^{ème}$ valeur, etc. Si le pointeur i pointe sur la 1ère valeur de la 3ème ligne de la matrice, alors ce pointeur incrémenté de 4 pointera sur la 2ème valeur de la 3ème ligne de la matrice. De nouveau incrémenté de 4, il pointera sur la 3ème valeur de la 3ème ligne de la matrice, etc. On voit donc, qu'en faisant correspondre l'intervalle de stockage et le nombre de lignes, le pointeur accède aux éléments d'une ligne particulière.

### III.2.5 Calculs sur GPU

Une fois les données transférées, on peut faire le calcul sur la carte. Pour bien comprendre le fonctionnement de CUBLAS, on n'additionnera que la 1ère ligne de la première matrice avec la 1ère ligne de la 2ème matrice, c'est-à-dire qu'on fera une addition sélective sur les données.

Certaines fonctions CUBLAS permettent ce type d'opération lorsque les arguments passés sont correctement choisis. Les fonctions CUBLAS exécutant des fonctions standard, il n'existe pas toujours de fonction adaptée exactement aux calculs qu'on veut leur faire faire. On doit donc :

- prendre la fonction CUBLAS la plus proche du calcul qu'on veut faire exécuter,
- adapter les arguments de cette fonction CUBLAS au calcul à exécuter.

Pour faire cette addition, la fonction CUBLAS la plus approchante est la fonction `cublasSaxpy` qui réalise la fonction $y = \alpha * x + y$ et dont le prototype est :

```
cublasSaxpy(N, 1.0, dx, M, dy, M) ;
```

Si on l'applique à notre exemple, on réalise donc l'opération $dy = 1.0 * dx + dy$ avec $\alpha = 1$ ($2^{ème}$ argument).

On fixe l'intervalle de stockage à M (nombre de ligne) pour les deux matrices pour pouvoir accéder toujours à la $1^{ère}$ valeur de chaque colonne dans chacune des matrices.

La Figure 38 illustre le fonctionnement de la fonction `cublasSaxpy()`.

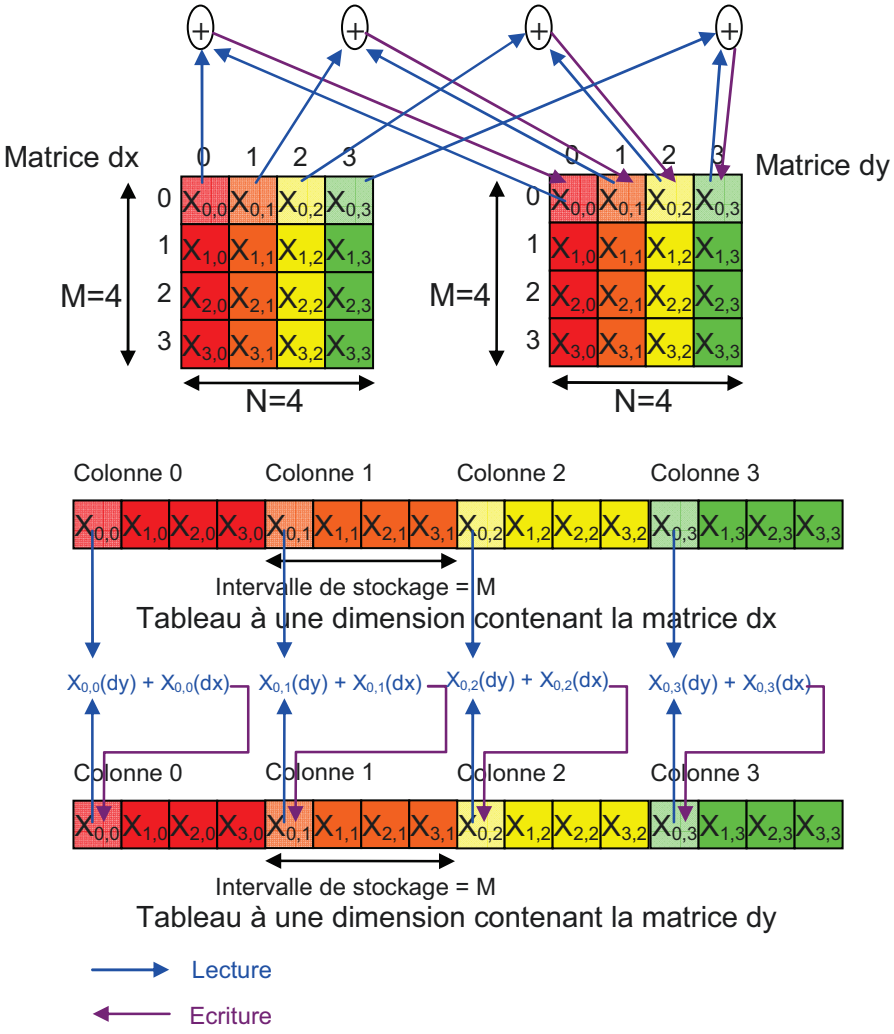



Figure 38 : Fonctionnement de la fonction cublasSaxpy

Pour i variant de *0* à *N-1*, cette fonction exécute l'opération $\alpha * (dx[i * M]) + (dy[i * M])$ avec $\alpha=1$ et place le résultat dans $dy[i * M]$

On voit bien qu'en ayant accédé au $1^{er}$ élément de chaque colonne, c'est-à-dire en se déplaçant par intervalles de M valeurs dans le tableau à une dimension, on a accédé globalement à l'ensemble des valeurs de la $1^{ère}$ ligne. Autrement dit, on a accédé à des valeurs non consécutives de chacun des deux tableaux à une dimension contenant les matrices.

### III.2.6 Transfert des données GPU vers CPU

Une fois le calcul réalisé sur la carte graphique, il ne reste qu'à le transférer sur le CPU. On utilise pour cela la fonction `cublasGetVector()` dans l'instruction :

```
cublasGetVector(M*N, sizeof(double), dy, 1, z, 1);
```

Cette fonction fonctionne de manière similaire à la fonction `cublasSetVector()`. Pour `cublasgetVector()`, les données sont copiées du GPU (`dy`) au CPU (`z`).

### III.2.7 Libération de la mémoire GPU

Une fois le résultat transféré sur le CPU, la bibliothèque n'étant plus utilisée, on doit libérer les espaces mémoire de la carte graphique avec les instructions :

```
cublasFree(dx);
cublasFree(dy);
```

On doit ensuite libérer les ressources côté CPU par l'instruction :

```
cublasShutdown();
```

Dès que le résultat est transféré sur CPU, on peut l'utiliser dans des instructions C ordinaires : affichage, calculs non parallélisables.

### III.2.8 Libération de la mémoire CPU

On libère ensuite la mémoire CPU avec la fonction C habituelle `free()`.

## III.3 Application de CUBLAS à la multiplication de matrices dans une fonction MEX

Pour pouvoir intégrer l'utilisation de CUBLAS dans une fonction MEX dans le but d'exécuter une multiplication de matrices, il suffit d'utiliser les mêmes fonctions MEX que celles déjà décrite dans la partie CUDA de ce rapport. De la même manière que pour le programme CUBLAS de base, on utilise de nouveau les fonctions :

- `cublasInit()` et `cublasShutdown()` pour ouvrir et fermer la bibliothèque,
- `cublasAlloc()` et `cublasFree()` pour réserver et libérer de l'espace mémoire GPU,
- `cublasSetVector()` et `cublasGetVector()` pour transférer les données (matrices) CPU vers GPU et GPU vers CPU.

Toutes les fonctions CUBLAS qui ne font pas de calcul, c'est-à-dire les fonctions

- de transfert,
- de préparation des calculs,
- de libération des ressources.

renvoient une variable d'état prédéfinie intitulée `cublasStatus`. Cette variable contient l'état de la fonction concernée.

L'état permet de signaler que la fonction :

- soit s'est exécuté normalement,
- soit ne s'est pas exécutée normalement. Dans ce cas le type d'erreur est renvoyé dans la variable `cublasStatus`.

Les fonctions de calcul ne renvoient pas de variable d'état. Cependant, on peut l'obtenir en utilisant la fonction `cublasGetError()`.

Par exemple, pour la fonction `cublasInit()`, on aura le code suivant permettant de tester que cette fonction s'est déroulée correctement. Dans le cas contraire, on affiche un message d'erreur :

```
if (status != CUBLAS_STATUS_SUCCESS) {
 printf("Erreur de transfert des données");
 cublasFree(dx);
 cublasShutdown();
 return EXIT_FAILURE;
 }
```

Les vecteurs opérandes (vecteurs d'*entrée*) n'ont pas besoin d'être initialisés puisqu'ils reçoivent les valeurs transmises par la fonction MEX. Seul le vecteur de retour est alloué de la même manière que dans le programme CUDA étudié précédemment.

On peut aussi afficher un message d'erreur à l'aide d'une fonction MEX de la manière suivante :

```
mexErrMsgTxt("Erreur d'accès au périphérique lors de l'écriture de la matrice A\n");
```

Une fois les données transférées de la même manière que dans le programme de base, on utilise la fonction de calcul `cublasGemm()` qui réalise en intégralité la multiplication des deux matrices d'entrée grâce à l'instruction :

```
cublasDgemm('n', 'n', mrowsA, ncolsB, ncolsA, alpha, d_A, mrowsA, d_B, mrowsB, beta, d_C, mrowsA);
```

La fonction `cublasDgemm()` fait partie des fonctions de la liste BLAS3 de la bibliothèque, c'est-à-dire que ses opérandes sont deux matrices. Elle a été choisie parmi les fonctions BLAS3 opérant sur des nombres réels double précision car c'est ce qui correspond au format utilisé par MATLAB. Des variantes de cette fonction permettent également de faire le même calcul sur des nombres :

- complexes double précision,
- réels simple précision,
- réels double précision.

On peut résumer les opérations matrice/matrice qu'exécute cette fonction sur la carte graphique par la formule suivante :

$$d\_C = (\alpha * d\_A * d\_B) + (\beta * d\_C)$$

Avec :

- $\alpha$ et $\beta$ : nombres flottants double précision,
- `d_A`, `d_B` et `d_C` : matrices GPU constituées d'éléments double précision, avec :
  - `d_A` : matrice GPU de taille $mrows1 * ncols1$,
  - `d_B` : matrice GPU de taille $ncols1 * ncols2$,
  - `d_C` : matrice GPU de taille $mrows1 * ncols2$,
- `mrowsA` : nombre de lignes de la matrice `d_A`,
- `ncolsA` : nombre de colonnes de la matrice `d_A`,
- `mrowsB` : nombre de lignes de la matrice `d_B`,
- `ncolsB` : nombre de colonnes de la matrice `d_B`,
- `n` : paramètre qui signifie « pas de transposée » (*non-transpose*) sur les matrices opérandes.

Une fois qu'on comprend l'utilisation d'une fonction CUBLAS, l'utilisation des autres fonctions de cette bibliothèque est relativement simple car similaire. A part quelques exceptions, la plupart des fonctions CUBLAS comporte les mêmes types d'arguments :

- choix ou non de calculs sur transposées,
- nombre de ligne/colonnes,
- pointeurs sur matrices d'entrée/sortie,
- dimensions dominantes,
- coefficients $\alpha$ et $\beta$.

## III.4 Analyse des résultats pour l'API CUBLAS

Comme pour la fonction MEX-CUDA-C, les fonctions complémentaires aux fonctions CUBLAS ont été choisies parmi les fonctions de l'interface de bas niveau de l'API du moteur d'exécution CUDA.

Contrairement au code CUDA dans lequel les tailles des blocs et le nombre de registres étaient fixés par l'utilisateur, dans un code CUBLAS, c'est l'API CUBLAS qui fixe le nombre de tâches par bloc et le nombre de registres par tâche au moment de la compilation. C'est pourquoi dans un programme CUBLAS, l'instruction ***-maxrregcount*** n'a aucune influence sur les résultats et la taille des blocs n'est pas configurable par l'utilisateur.

Le compilateur *nvcc* (voir §II.1) détermine ces paramètres à partir de la taille de la matrice résultat uniquement.

Comme pour CUDA, la communication entre MATLAB et le programme se fait via une fonction MEX. Comme précédemment, les tests ont été réalisés en mode Release. Les résultats ont été moyennés sur 10 valeurs avec une exécution préalable non prise en compte car due aux incertitudes des temps de calcul lors de la mise en place des ressources de la carte graphique. Les FLOPS ont été calculé de la même façon que pour CUDA, c'est-à-dire : $FLOPS = \frac{2 * n^3}{tpscalcul}$

La procédure de vérification des résultats de calcul et de mesure des temps d'exécution a été réalisée de manière similaire à celle de CUDA (voir §II.5).

Contrairement au programme CUDA, un programme utilisant l'API CUBLAS ne nécessite pas d'avoir des tailles de matrice résultat multiples de la taille des blocs. Une taille de matrice résultat non multiple de la taille de bloc donnera des résultats exacts pour un programme MEX-CUBLAS-C, alors qu'ils auraient été faux pour un programme MEX-CUDA-C seul.

D'après les mesures réalisées, on remarque que CUBLAS choisit certains paramètres à partir de la taille de la matrice résultat. Les premiers essais réalisés ont montré des fluctuations de résultat importantes pour des tailles de matrice résultat proches (par exemple des matrice résultat de taille *1024 * 1024* et *1020 * 1020*). La décomposition en facteurs premiers de la taille des matrices résultat a une influence sur l'efficacité du calcul. C'est pourquoi les tests qui ont suivi ont été menés dans différents cas de figure en tenant compte des facteurs premiers.

Evidemment les mesures pour des tailles de matrice résultat multiples de 64 incluent également les mesures pour des tailles de matrice résultat multiples de 128 puisque 128 est multiple de 64. Le même raisonnement est généralisable aux multiples de 32, 16, 8 et 4. Donc la figure pour des multiples de 4 contient également les mesures pour tous les multiples supérieurs à 4.

Les mesures étant nombreuses, elles sont représentées sous forme de courbes pour plus de commodité.

### III.4.1 Influence de la taille de la grille de tâches sur la performance

Les courbes des figures 39 à 44 illustrent les mesures de GFLOPS (à partir des temps d'exécution) pour des grilles de tâches dont chaque côté est multiple de (respectivement) 64, 32, 16, 14, 8 et 4.
Dans les courbes suivantes :

- Les courbes « MEX-CUBLAS-C » correspondent au nombre de FLOPS déduits de la mesure du temps séparant les instants suivants :
  - appel sous MATLAB de la fonction MEX-CUBLAS-C,
  - fin de la réception par MATLAB des résultats du programme.

  Les FLOPS représentés sur les courbes « MEX-CUBLAS-C » correspondent donc à des temps incluant les temps de transfert entre CPU et GPU.
- Les courbes intitulées « Kernel » représentent les FLOPS déduits de la mesure des temps d'exécution du calcul par la carte graphique seule. Ils n'incluent donc pas les temps de transfert des données.
- Les courbes intitulées « MATLAB » représentent les FLOPS déduits de la mesure des temps d'exécution du calcul par MATLAB seul (sans le programme MEX-CUBLAS-C).

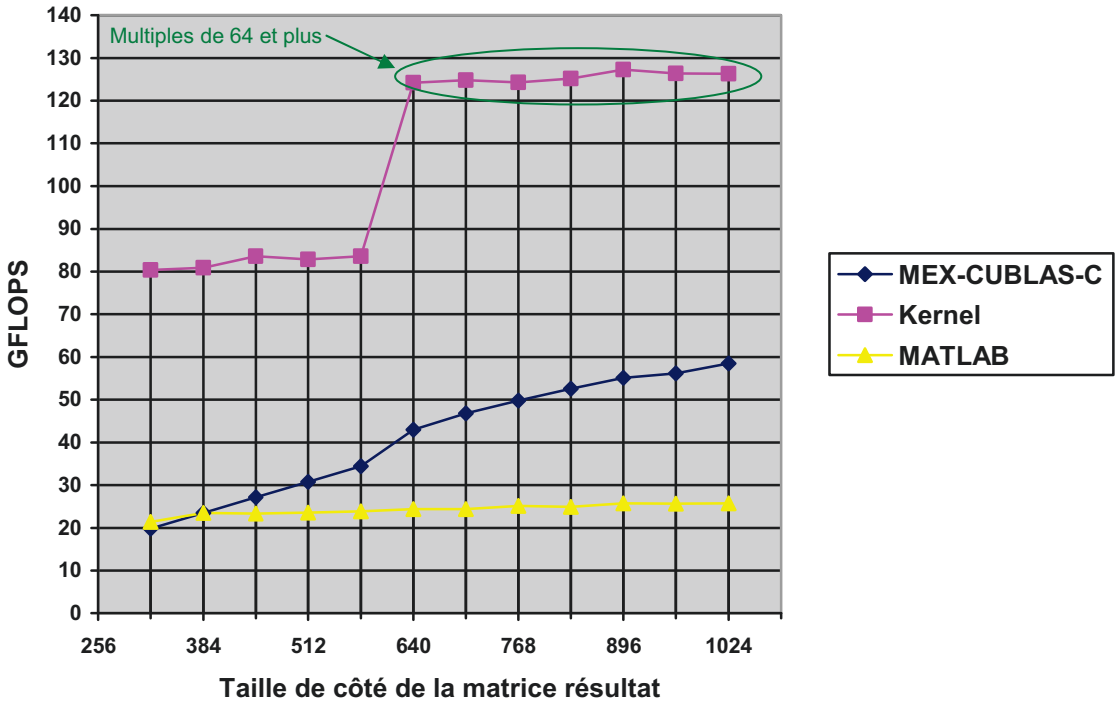



Figure 39 : GFLOPS pour des côtés de matrice résultat multiples de 64

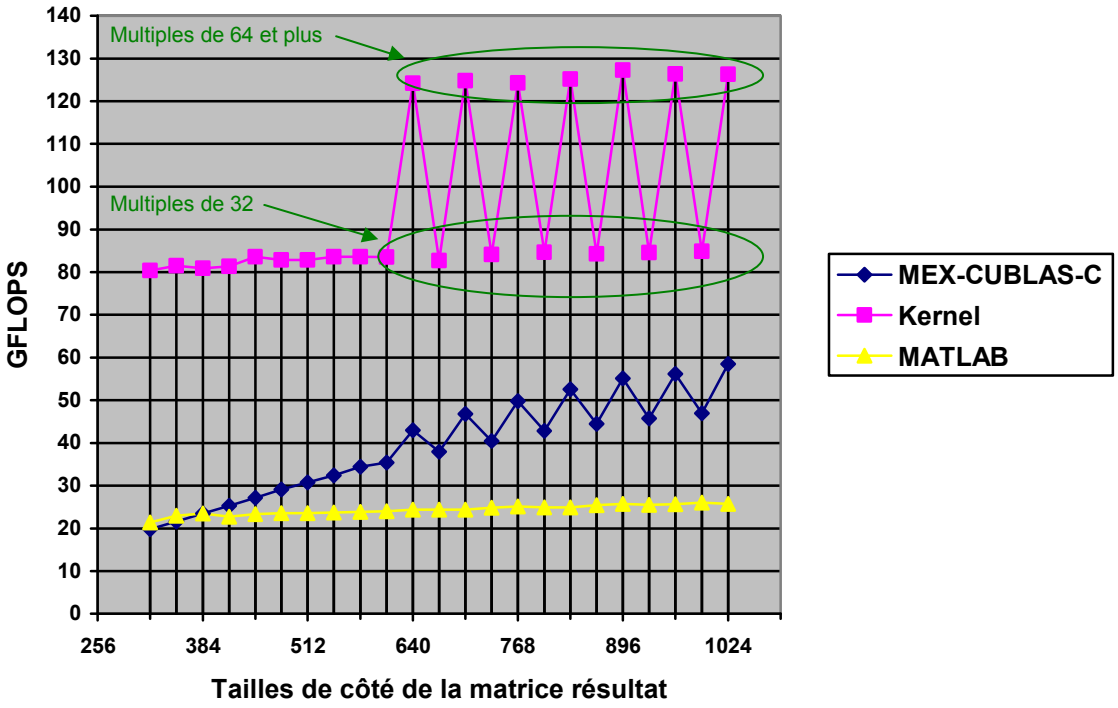



Figure 40 : GFLOPS pour des côtés de matrice résultat multiples de 32

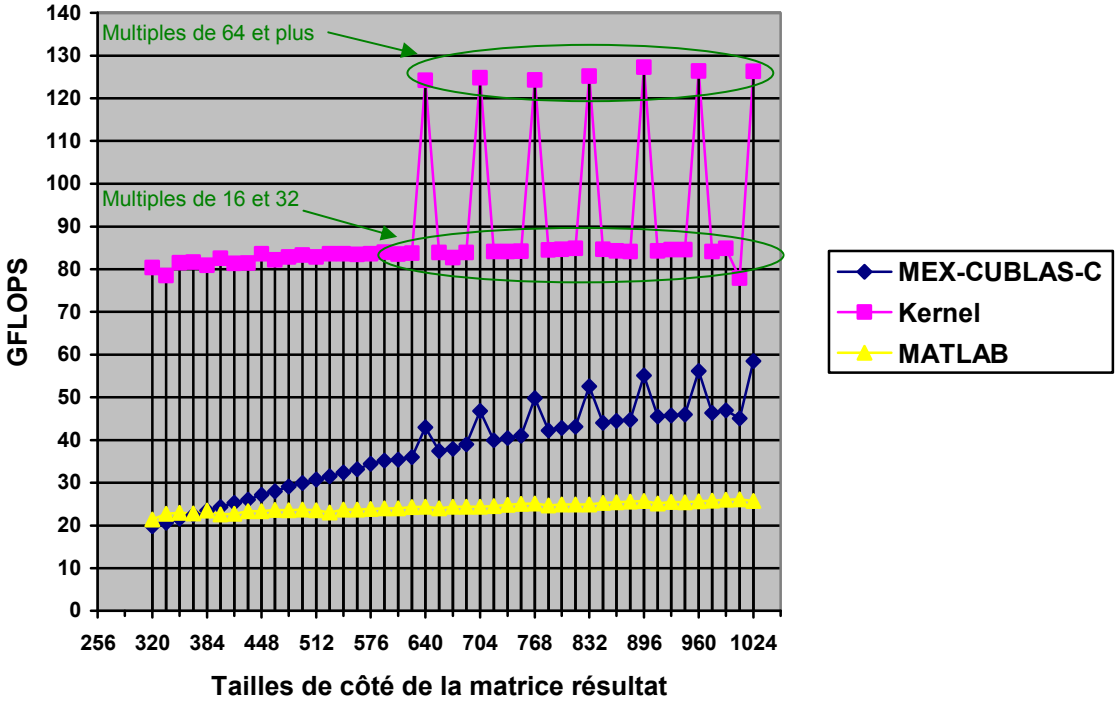



Figure 41 : GFLOPS pour des côtés de matrice résultat multiples de 16

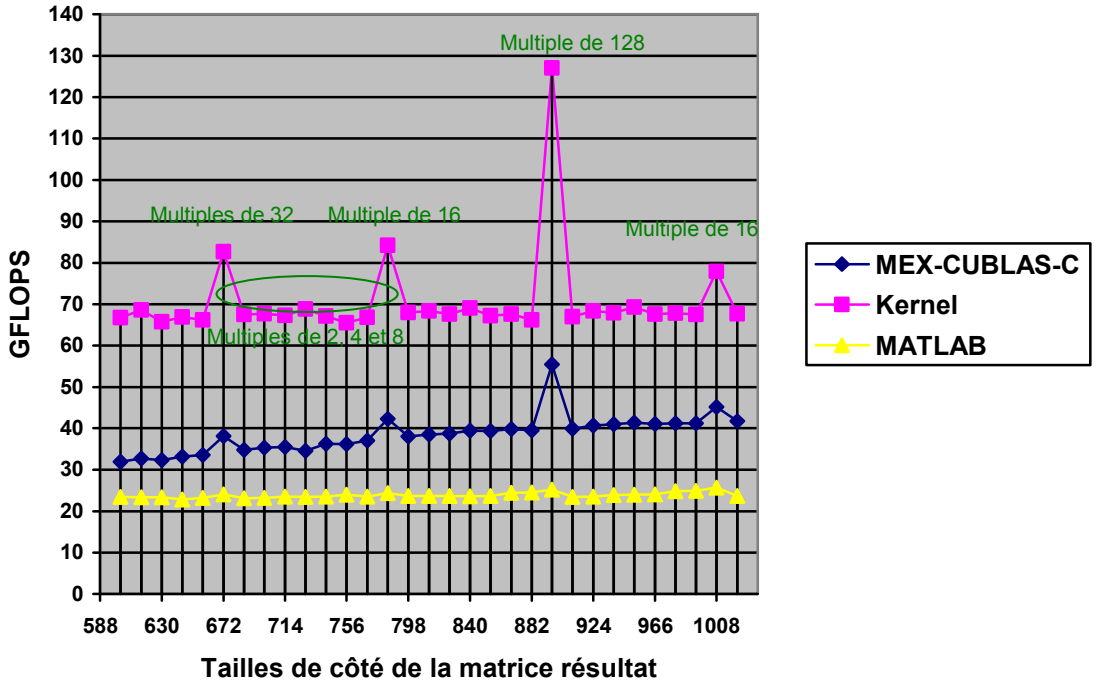



Figure 42 : GFLOPS pour des côtés de matrice résultat multiples de 14 (nombre de multiprocesseurs de la carte GTX470)

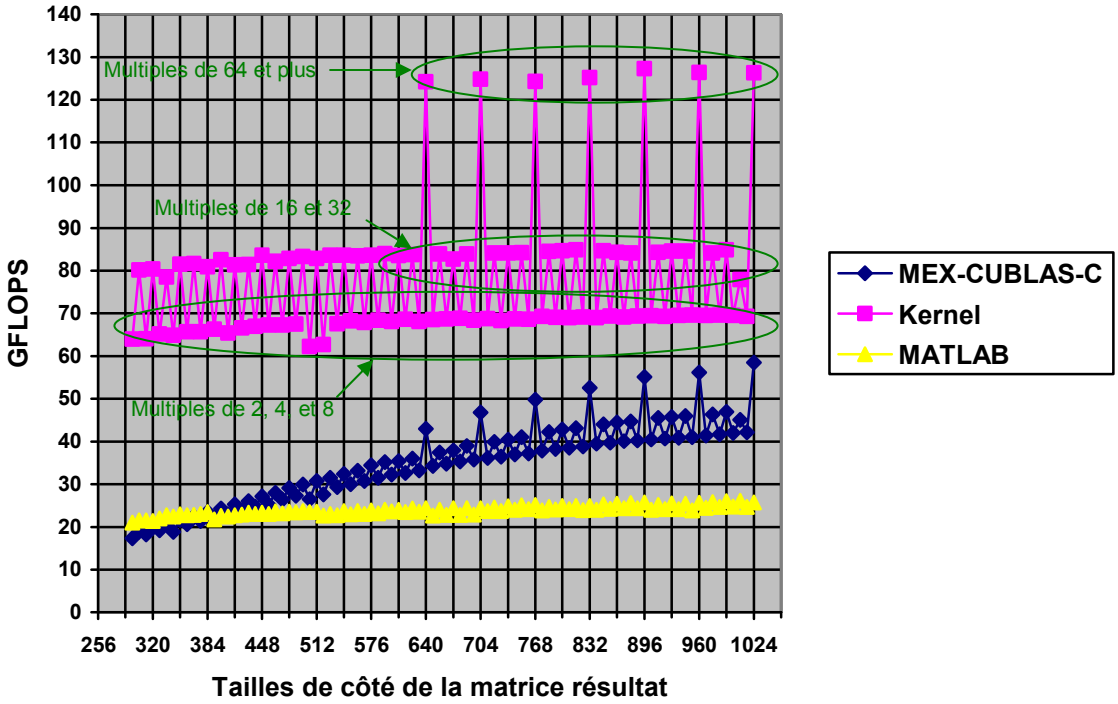



Figure 43 : GFLOPS pour des côtés de matrice résultat multiples de 8

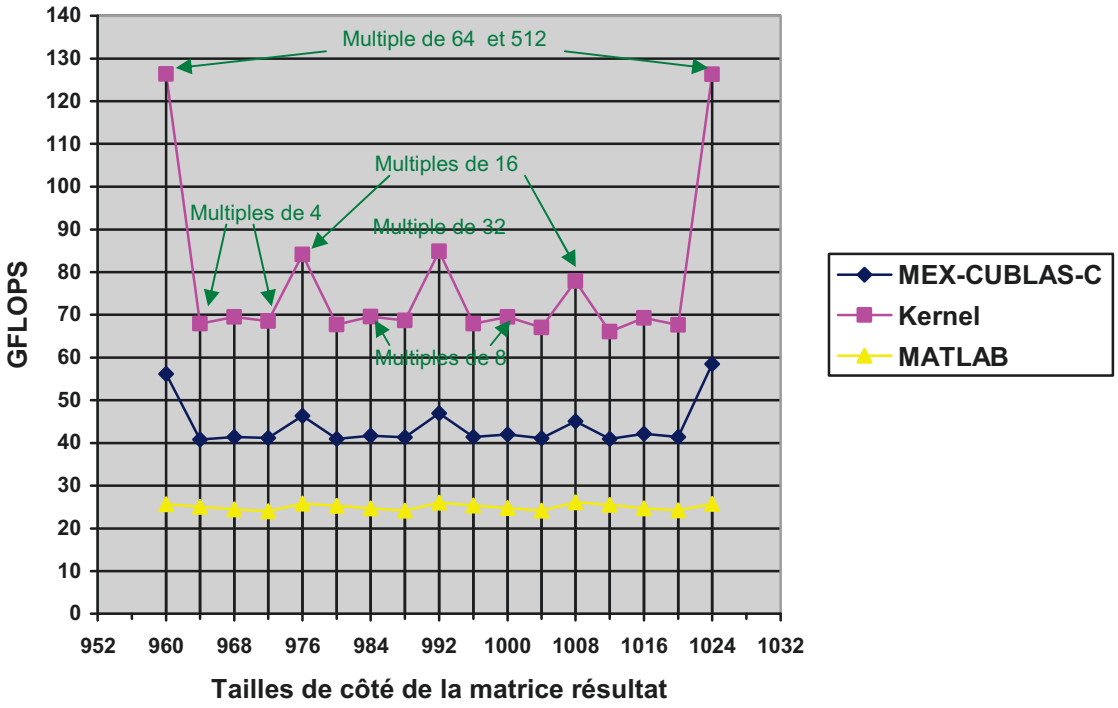



Figure 44 : GFLOPS pour des côtés de la matrice résultat multiples de 4

Le multiple de la taille de côté de la matrice résultat a une certaine importance. On compare pour des tailles de matrices résultat comparables la puissance de calcul.

On remarque que globalement, plus la taille de la matrice résultat est multiple d'une puissance de 2 élevée, plus la performance est bonne. Par contre, il faut éviter d'avoir des tailles de côté inférieures à 16, car ce serait sous-exploiter les capacités du processeur graphique.

Sur la Figure 42, on remarque qu'avoir une taille de côté de matrice opérande multiple de 14 (nombre de multiprocesseurs de la carte graphique GTX470) n'apporte pas d'amélioration significative au niveau du nombre de FLOPS, au contraire. Les matrices résultat dont les côtés sont multiples de 14 donnent des résultats très proches de celles dont les côtés sont multiples de 8 ou moins.

Pour la fonction MEX-CUBLAS-C, les différences de performance sont moins visibles car elles sont en partie masquées par :

- les temps de transfert de données entre MATLAB et la fonction (de l'ordre de 15 ms au total),
- les temps de transfert des données entre CPU et GPU (non mesurés).

D'autre part, on remarque que la fonction MEX-CUBLAS-C reste plus performante que MATLAB jusqu'à des tailles de matrice résultat relativement faibles ($400 \times 400$). Pour des côtés de matrice inférieurs à 400, MATLAB gagne en performance par rapport à la fonction MEX- CUBLAS-C.

### III.4.2 Influence de la taille de la grille de tâches sur le choix des paramètres par CUBLAS

On rappelle que pour CUBLAS, la plupart des paramètres sont gérés (choisis) par le gestionnaire de tâches du processeur graphique. Pour chaque mesure des figures 39 à 44, un certain nombre de ces paramètres ainsi que d'autres ont été observés grace au profiler, à savoir :

- le nombre de registres utilisés par tâche,
- la taille de grille suivant ses 3 dimensions,
- la taille de bloc suivant ses 3 dimensions,
- la quantité de mémoire partagée utilisée (en octets),
- le nombre de blocs résidents par multiprocesseur,
- le taux d'occupation.

A l'aide du profiler, on remarque que CUBLAS n'utilise que deux configurations selon la taille de matrice résultat.

On observe que :

- dans le cas où on a des côtés de matrice résultat de taille **supérieure ou égale à 640 et multiple de 64 ou plus**, le gestionnaire de tâches choisit toujours les mêmes valeurs de paramètres (quelle que soit la taille de la matrice résultat), à savoir :
  - 52 registres par tâche,
  - blocs non carrés de 16 lignes et 4 colonnes,
  - 2176 octets de mémoire partagée par bloc,
  - 8 blocs résidents par multiprocesseur,

  résultant ainsi en un taux d'occupation de 33%.
- dans tous les autres cas, le gestionnaire de tâches choisit toujours les mêmes valeurs de paramètres (quelle que soit la taille de la matrice résultat), à savoir :
  - 20 registres par tâche,
  - blocs non carrés de 256 lignes et 1 colonne,
  - 4352 octets de mémoire partagée par bloc,
  - 6 blocs résidents par multiprocesseur,

  résultant ainsi en un taux d'occupation de 100%.

La manière dont l'API CUBLAS choisit les paramètres qu'elle utilise n'est pas décrite dans la documentation fournie par NVIDIA. L'étude des résultats peut nous amener à émettre une hypothèse probable de fonctionnement de CUBLAS concernant le choix des paramètres.

On se rappelle (§II.4.2.7) que le gestionnaire de tâches vérifie que les ressources nécessaires à un bloc de tâches sont présentes dans le multiprocesseur ciblé. Si ce n'est pas le cas, le gestionnaire diminue le nombre de blocs résidents à attribuer jusqu'à ce que les ressources nécessaires soient inférieures ou égale aux ressources matérielles du multiprocesseur.

On a vu précédemment (§II.4.2.7), que pour vérifier que les ressources matérielles sont suffisantes, le gestionnaire de tâches vérifie que les conditions suivantes sont respectées :

- 32768 registres maximum par multiprocesseur,
- 1536 tâches résidentes maximum par multiprocesseur,
- 48 ko maximum de mémoire partagée par multiprocesseur,
- 8 blocs résidents maximum par multiprocesseur,

- 16 registres par tâche minimum.

Une hypothèse de fonctionnement de CUBLAS est décrite sur la Figure 45.

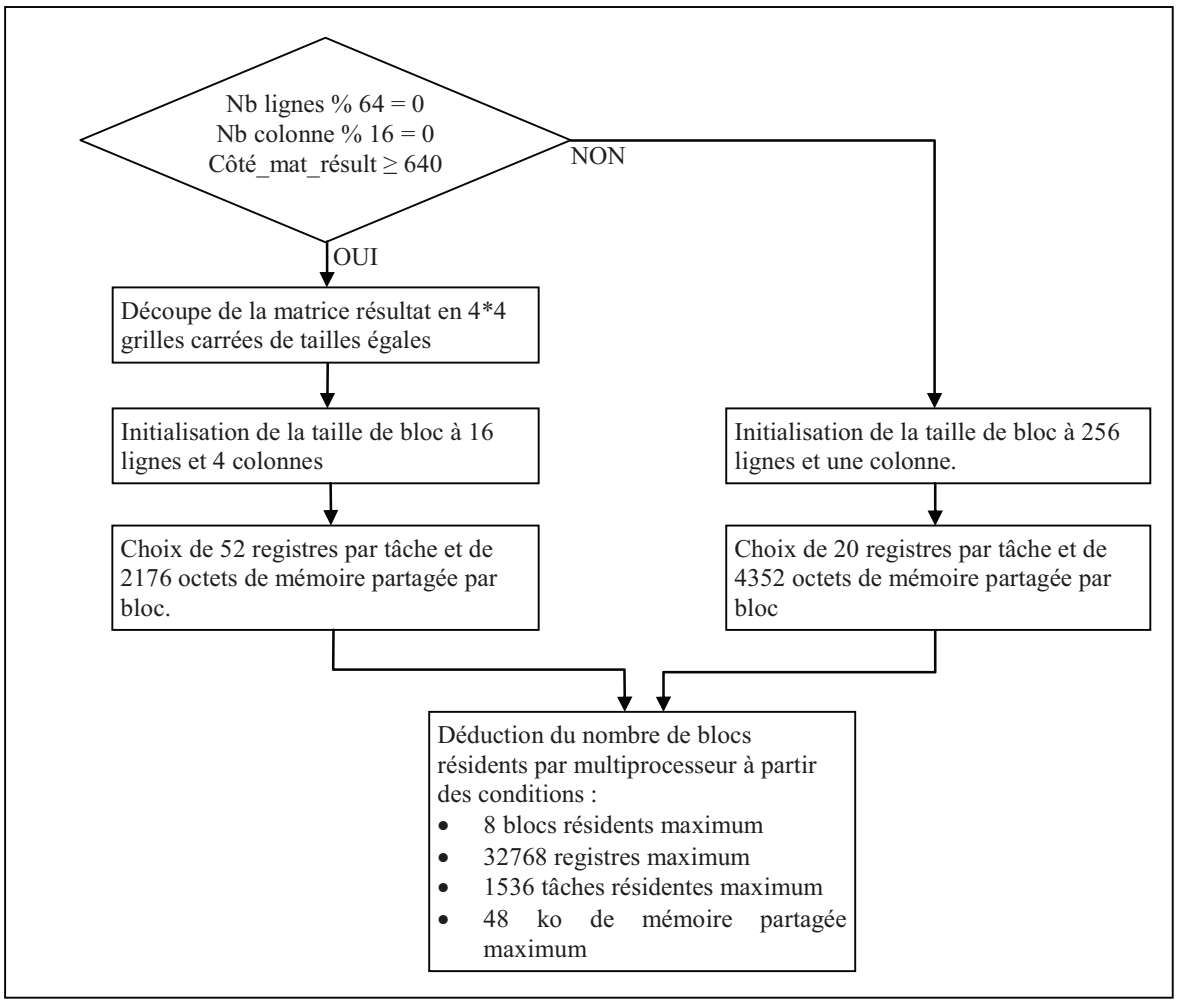



Figure 45 : Hypothèse de fonctionnement de CUBLAS concernant le choix des paramètres

Prenons deux exemples pour illustrer la procédure décrite sur la Figure 45, un exemple avec une taille de matrice résultat de :

- 656*656
- 640*640

### III.4.2.1 Cas d'une matrice résultat de taille 656*656 :

656 est supérieur à 640 et mais n'est pas divisible par 64. Le gestionnaire de tâches décide alors :

- que chaque bloc comportera 256 lignes et une colonne,
- qu'il y aura une seule grille de la taille de la matrice résultat.
- que 20 registres par tâches seront utilisés,
- que 4352 octets de mémoire partagée par bloc seront utilisés.

Le gestionnaire de tâches déduit alors le nombre de blocs à attribuer par multiprocesseur (futurs blocs résidents) à partir des choix précédents et des conditions :

- 32768 registres maximum,
- 1536 tâches résidentes maximum,
- 48 ko de mémoire partagée maximum,

- 8 blocs résidents maximum.

Chaque tâche nécessite 20 registres, donc un bloc de 256 tâches nécessite 5120 registres. 8 blocs de 256 tâches nécessitent 40960 registres. La 1ère condition n'est pas respectée. Pour que cette condition soit respectée, le gestionnaire de tâches devra attribuer 6 blocs résidents maximum car 6 blocs nécessitent 30720 registres (inférieur au maximum de 32768 registres).

Chaque bloc comporte 256 tâches. 8 blocs comportent donc 2048 tâches. La 2ème condition n'est pas respectée. Pour que cette condition soit respectée, le gestionnaire de tâches devra attribuer 6 blocs résidents maximum car 6 blocs comportent 1536 tâches.

Chaque bloc nécessite 4352 octets de mémoire partagée par bloc. 8 blocs nécessiteront donc 34816 octets, ce qui satisfait la 3ème condition.

Le nombre de blocs résidents est fixé à 6 par le gestionnaire de tâches, ce qui permet de satisfaire :

- La condition 1 : 20 registres/tâche * 256 tâches/bloc * 6 blocs = 30720 < 32768
- La condition 2 : 256 tâches/bloc * 6 blocs résidents = 1536 tâches
- La condition 3 : 4352 octets/bloc * 6 blocs résidents = 25,5 ko < 48 ko
- La condition 4 : 6 blocs résidents < 8 blocs résidents maximum

On peut en déduire le taux d'occupation des ressources, c'est-à-dire le rapport du nombre de warps résidents par le nombre maximal de warps résidents.

Chaque bloc contient 256 tâches, et un warp contient 32 tâches, on aura donc $\frac{256}{32} = 8$ warps résidents par bloc. Par conséquent le nombre de warps résidents contenus dans les 6 blocs résidents sera de $8 * 6 = 48$. Le nombre maximal de warps résidents par multiprocesseur étant de 48, on en déduit le taux d'occupation : $\frac{48}{48} = 1$ c'est-à-dire 100%. On a donc 100% d'occupation des ressources.

### III.4.2.2 Cas d'une matrice résultat de taille 640*640 :

Dans ce cas, 640 est divisible par 64, l'ensemble des tâches associées à la matrice résultat est donc divisé en 16 grilles de taille 160*160.

Le gestionnaire de tâches décide ensuite d'attribuer :

- une taille de 16 lignes et 4 colonnes par bloc,
- 52 registres par tâches,
- 2176 octets de mémoire partagée par bloc.

Le gestionnaire de tâches calcule alors le nombre de blocs à attribuer par multiprocesseur (futurs blocs résidents) à partir des choix précédents et des conditions :

- 32768 registres maximum,
- 1536 tâches résidentes maximum,
- 48 ko de mémoire partagée maximum,
- 8 blocs résidents maximum.

Le nombre de blocs résidents est fixé à 8 par le gestionnaire de tâches, ce qui permet de satisfaire :

- La condition 1 : 52 registres/tâche * 64 tâches/bloc * 8 blocs = 26624 < 32768,
- La condition 2 : 64 tâches/bloc * 8 blocs résidents = 512 tâches < 1536,
- La condition 3 : 2176 octets/bloc * 8 blocs résidents = 17 ko < 48 ko,

- La condition 4 : 8 blocs résidents = 8 blocs résidents maximum.
  On peut en déduire le taux d'occupation des ressources.

Chaque bloc contient 64 tâches, et un warps contient 32 tâches, on aura donc $\frac{64}{32} = 2$ warps résidents par bloc. Par conséquent le nombre de warps résidents contenus dans les 8 blocs résidents sera de $2 * 8 = 16$. Le nombre maximal de warps résidents par multiprocesseur étant de 48, on en déduit le taux d'occupation : $\frac{16}{48} = 0{,}33$ c'est-à-dire 33%. On a donc 33% d'occupation des ressources.

### III.4.3 Synthèse des résultats obtenus

Les résultats peuvent être classés selon les cas dans le Tableau 12.

Tableau 12 : Caractéristiques des deux modes de fonctionnement de CUBLAS selon la taille de matrice résultat

| Côté matrice résultat ≥ 640<br>Nb lignes résultat % 64 = 0<br>Nb colonnes résultat % 16 = 0 | Non<br>(exemple : 656 * 656) | Oui<br>(exemple 640 * 640) |
|---|---|---|
| Nb de grilles de tâches | 1 | 16 |
| Taille de grille | Taille de la matrice résultat | 160*160 |
| Nb lignes par bloc | 256 | 16 |
| Nb colonnes par bloc | 1 | 4 |
| Nb tâches par bloc | 256 | 64 |
| Nb de registres par tâches | 20 | 52 |
| Nb octets de mémoire partagée par bloc | 4352 | 2176 |
| Nb blocs résidents par multiprocesseur | 6 | 8 |
| Nb warps résidents par bloc | 8 | 2 |
| Nb warps résidents par multiprocesseur | 48 | 16 |
| Taux d'occupation | 100% | 33% |
| Nb GFLOPS | 37,4 | 43,0 |

D'après le Tableau 12, on remarque que contrairement à un programme MEX-CUDA-C, la performance d'un programmeMEX-CUBLAS-C n'est pas liée au taux d'occupation, ni au nombre de tâches par bloc mais dépend avant tout du nombre de blocs résidents par multiprocesseur.

D'après ces résultats et d'après les figures précédentes, on peut en déduire que pour avoir de bonnes performances avec CUBLAS, on doit s'arranger pour avoir une taille de matrice résultat qui entraîne un nombre de blocs résidents maximal, quitte à diminuer le taux d'occupation.

Sur le Tableau 13, on observe les temps d'exécution et de transfert de la fonction MEX-CUBLAS-C. On remarque que plus la taille de la matrice résultat est grande, plus la proportion de temps de transfert est faible, contrairement à CUDA pour lequel cette proportion de temps de transfert augmente.

Tableau 13 : Temps de transfert et d'exécution de la fonction MEX-CUBLAS-C

| Taille de la matrice résultat | Fonction MEX-CUBLAS-C | | | | | |
|---|---|---|---|---|---|---|
| | Temps d'exécution (ms) | Kernel seul (ms) | Temps de transfert CPU → GPU (ms) | Temps de transfert GPU → CPU (ms) | Temps total (kernel + transferts) (ms) | Proportion temps transfert / temps total (%) |
| 1024*1024 | 36,7 | 17,0 | 4,06+3,51 | 3,08 | 27,7 | 29,0 |
| 960*960 | 31,5 | 14,0 | 4,41+3,25 | 3,11 | 24,8 | 34,2 |
| 896*896 | 26,1 | 11,3 | 3,72+2,93 | 2,76 | 20,7 | 36,1 |

# IV Utilisation de la bibliothèque CUFFT pour le traitement du signal

## IV.1 Présentation de la bibliothèque CUFFT

La bibliothèque CUFFT est la bibliothèque de transformées de Fourier rapide (FFT) de CUDA™ [22]. Cette bibliothèque offre une interface simple pour traiter informatiquement des FFT sur un processeur graphique NVIDIA, ce qui permet d'exploiter le parallélisme du processeur graphique sans avoir à développer une mise en œuvre FFT basée sur le processeur graphique.

Cette version de bibliothèque met en œuvre les caractéristiques suivantes :

- transformées 1D, 2D, et 3D de données réelles ou complexes,
- possibilité de regrouper des FFT de différentes tailles et de les exécuter simultanément (en parallèle),
- tailles de transformées comprenant
  - jusqu'à 64 millions d'éléments simple précision,
  - jusqu'à 128 millions d'éléments double précision

  dans n'importe quelle dimension,
- transformées
  - « sur-place » (donnée d'entrée remplacée par le résultat),
  - dans une variable de résultats

  pour des données réelles et complexes,
- transformées double précision sur matériel compatible (processeurs de type GT200 et suivants).

Cette bibliothèque est constituée de deux parties principales :

- types CUFFT prédéfinis et définitions de direction de transformée
- fonctions CUFFT prédéfinies préparant les transformées et les exécutant.

L'API CUFFT est basée sur la bibliothèque FFTW, qui est une des bibliothèques FFT les plus populaires et les plus efficaces. La bibliothèque FFTW offre un mécanisme de configuration simple appelé *plan* qui spécifie complètement le plan d'exécution optimal – c'est-à-dire le nombre de d'opérations en virgule flottante (FLOP) minimal – pour une taille de FFT et un type de donnée particuliers.

L'avantage de cette approche est que une fois que l'utilisateur a créé un plan, la bibliothèque stocke tout état nécessaire pour exécuter le plan de nombreuses fois sans recalcul de la configuration. Le modèle FFTW fonctionne bien pour la bibliothèque CUFFT parce que différents types de FFT demandent des ressources GPU et des configurations de tâche différentes, et les plans sont des moyens simples de stocker et réutiliser des configurations.

La bibliothèque CUFFT initialise les données internes au moment du premier appel d'une fonction de l'API. Par conséquent, toute fonction de l'API peut renvoyer le code d'erreur CUFFT_SETUP_FAILED si l'initialisation de la bibliothèque a échoué. Cette bibliothèque se ferme automatiquement lorsque tous les plans de transformée de Fourier créés par l'utilisateur ont été détruits.

## IV.2 Description d'un programme CUFFT de base

Un programme de base utilisant la bibliothèque CUFFT est constitué des parties suivantes :

- déclaration des variables et des pointeurs CPU et GPU,
- allocation d'espace mémoire sur la carte graphique et sur l'hôte,
- initialisation et transfert des données opérandes sur GPU,
- création d'un plan de transformée contenant les caractéristiques principales de la transformée souhaitée,
- exécution de la transformée de Fourier,
- transfert du résultat sur CPU,
- destruction du plan de transformée,
- libération des espaces mémoire CPU et GPU.

Comme programme CUFFT de base, nous étudierons la transformée de Fourier rapide d'un vecteur de 10 valeurs opérandes (également appelées valeurs *d'entrée*). Pour faciliter les calculs de transformée de Fourier rapide, la bibliothèque CUFFT comporte des types de donnée prédéfinis.

Dans ce programme, nous utiliserons le type `cufftComplex` qui permet de manipuler des données complexes simple précision. On utilise ici des données simple précision uniquement pour réduire le programme à sa plus simple expression. Après avoir déclaré des pointeurs, on leur alloue de l'espace mémoire avec les instructions :

```
idata = (cufftComplex*)malloc(sizeof(cufftComplex)*NX*BATCH);
odata = (cufftComplex*)malloc(sizeof(cufftComplex)*NX*BATCH);
```

avec :

- `NX` : taille de la transformée,
- `BATCH` : nombre de transformées à exécuter simultanément.

Une fois la mémoire réservée sur l'hôte, on peut initialiser les 10 données opérandes et les 10 variables résultat (également appelée valeurs de *sortie*) dans une boucle de 10 tours.

Le type `cufftComplex` [22] est une structure de deux paramètres :

- `x` : partie réelle de la variable de ce type,
- `y` : partie imaginaire de la variable de ce type.

On initialise les données d'entrée avec des valeurs réelles et entières pour avoir un calcul facilement vérifiable à la main. On initialise donc ces données de manière arbitraire à (par exemple) 2 fois leur indice de tableau avec les instructions :

```
idata[i].x = 2*i; /* Partie réelle */
idata[i].y = 2*i; /* Partie imaginaire */
```

On initialise les données de sortie à `0` de manière similaire.

Avant de pouvoir transférer sur la carte graphique les données initialisées, on réserve de la mémoire pour le pointeur GPU `devPtr` grace à l'instruction :

```
cudaMalloc((void**)&devPtr, sizeof(cufftComplex)*NX*BATCH);
```

On transfère ensuite les données pointées par `idata` de l'hôte vers le processeur graphique (GPU). Pour cela on utilise la fonction de transfert habituelle `cudaMemcpy` (voir §II.2) dans l'instruction :

```
cudaMemcpy(devPtr, idata, sizeof(cufftComplex)*NX*BATCH,
cudaMemcpyHostToDevice);
```

Une fois les données stockées sur le processeur graphique, on peut exécuter la transformée de Fourier proprement dite. Pour cela, on doit configurer la transformée à l'aide de la fonction `cufftPlan1d` dans l'instruction :

```
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);
```

avec :

- `plan` : variable déclarée comme un type prédéfini `cufftHandle`. Le type prédéfini `cufftHandle` est basé sur le type `unsigned int`. C'est un type utilisé pour stocker et accéder à des plans CUFFT.
- `&plan` : adresse du plan de configuration.
- `NX` : taille de la FFT.
- `BATCH` : nombre de transformées à exécuter simultanément.
- `CUFFT_C2C` : constante de type `cufftType` définissant le type de données manipulées par la transformée. `CUFFT_C2C` est une constante permettant de configurer une FFT en nombres flottants simple précision, avec données opérandes complexes et variables résultat complexes (*complex to complex*).

Une fois le plan de transformée configuré par la fonction `cufftPlan1d` , il ne reste plus qu'à exécuter la FFT grace à l'instruction :

```
cufftExecC2C(plan, devPtr, devPtr, CUFFT_FORWARD);
```

avec :

- `devptr` : pointeur GPU sur les données à traiter. Cette fonction exécute une FFT « sur place » (*in-place*), c'est-à-dire que le pointeur d'entrée est le même que celui de résultat.
- `CUFFT_FORWARD` : direction de la transformée (directe ou inverse), dans notre exemple on choisit volontairement une transformée directe pour simplifier le raisonnement [22].

Une fois que la transformée est executée, on renvoie le résultat au CPU. Pour cela on utilise la fonction de transfert habituelle `cudaMemcpy` (voir §II.2) dans l'instruction :

```
cudaMemcpy(odata, devPtr, sizeof(cufftComplex)* NX*BATCH, cudaMemcpyDeviceToHost);
```

Une fois le résultat du calcul renvoyé à l'hôte, si on ne souhaite pas exécuter d'autres transformées utilisant le même plan, on doit détruire le plan pour éviter de gaspiller la mémoire GPU. Pour détruire le plan, il existe la fonction `cufftDestroy` qui libère les ressources GPU associées à un plan CUFFT. On utilise cette fonction de la manière suivante :

```
cufftDestroy(plan);
```

On doit aussi libérer l'espace mémoire GPU avec l'instruction :

```
cudaFree(devPtr);
```

On libère également l'espace mémoire utilisé par les données d'entrée et de sortie sur l'hôte avec la fonction C habituelle `free`.

## IV.3 Application de la bibliothèque CUFFT au traitement du signal

Dans cette partie, nous étudierons un programme de transformée de Fourier directe appliquée à des données complexes double précision avec communication avec MATLAB via une fonction MEX. Nous n'étudierons que des transformées de nombres double précision car c'est le type manipulé par MATLAB par défaut.

Pour le programme principal, on utilise donc le nom `mexFunction` à la place de `main` comme cela a déjà été décrit précédemment dans la partie CUDA de ce mémoire.

Avant de recevoir les données venant de MATLAB, on déclare deux pointeurs hôte contenant la partie réelle et imaginaire des données à traiter :

```
double *prA, *piA;
```

avec :

`prA` : pointeur sur la partie réelle des données opérandes,
`piA` : pointeur sur la partie imaginaire des données opérandes.

On déclare également un pointeur GPU `devPtr` qui contiendra les données d'entrée/sortie de type `cufftDoubleComplex`.

MATLAB stocke et traite les matrices par colonne, de plus, la bibliothèque CUFFT stocke et traite les matrices également par colonne (dans un tableau), c'est pourquoi une transformée de Fourier par colonne sera exécutée. On initialisera donc :

- la taille de la transformée au nombre de lignes reçues de MATLAB,
- le nombre de transformées (à traiter simultanément) au nombre de colonnes de la matrice.

Pour accéder aux données transmises par MATLAB, on utilise les fonctions habituelles `mxGetM()`, `mxGetN()`, `mxGetPr()`, `mxGetPi` (similaire à `mxGetPr`).

Avant de rassembler la partie réelle et la partie imaginaire de chaque donnée en une donnée unique de type `cufftDoubleComplex`, on doit allouer de la mémoire pour les données d'entrée (`h_A`) et de sortie (`h_B`). Il existe une fonction intitulée `mxMalloc` qui :

- alloue dynamiquement de la mémoire en utilisant le gestionnaire de mémoire de MATLAB,
- renvoie un pointeur sur le début de la zone de mémoire allouée,
- libère la mémoire automatiquement lorsque le contrôle revient à MATLAB.

On peut appeler cette fonction de la manière suivante :

```
h_A = (cufftDoubleComplex*)mxMalloc(sizeof(cufftDoubleComplex) *
nx *  batch);
h_B = (cufftDoubleComplex*)mxMalloc(sizeof(cufftDoubleComplex) *
nx *  batch);
```

Une fois la mémoire allouée sur l'hôte, on rassemble la partie imaginaire et la partie réelle de chaque donnée venant de MATLAB en les convertissant en une variable de type `cufftDoubleComplex`. Cette fonction de conversion n'existe pas dans les fonction CUFFT disponibles. Par conséquent, on crée une fonction intitulée `pack_dc2dc` chargée de convertir en un nombre complexe les deux nombres séparés représentant les parties réelles et imaginaires de ce nombre.

Dans cette fonction, on fait une boucle comportant autant de tours que de variables de type `cufftDoubleComplex` à obtenir. Le type `cufftDoubleComplex` étant une structure de deux paramètres, il suffit de remplir chaque champ de la structure par la partie réelle et imaginaire de chaque donnée à chaque tour. La définition de la fonction `pack_dc2dc` est donc la suivante :

```
void pack_dc2dc(cufftDoubleComplex *output, double *input_re,
double *input_im, int Ntot)
{
int n;
for(n=0; n<Ntot; n++)
        {
        output[n].x = input_re[n];
        output[n].y =  input_im[n];
        }
}
```

On fait appel à cette fonction par l'instruction suivante :
`pack_dc2dc(h_A,prA,piA,nx*batch);`

avec :

- `h_A` : pointeur sur les données CPU converties en type `cufftDoubleComplex`,
- `prA` : pointeur sur la partie réelle des données venant de MATLAB,
- `piA` : pointeur sur la partie imaginaire des données venant de MATLAB,
- `nx*batch` : nombre de variables de type `cufftDoubleComplex` à obtenir.

Une fois les données converties en types `cufftDoubleComplex`, on les transfère avec la fonction `cudaMemcpy` après avoir alloué suffisamment de mémoire avec la fonction `cudaMalloc`.

On configure ensuite les paramètres de la transformée par l'instruction :
`cufftPlan1d(&plan, nx, CUFFT_Z2Z, batch);`

Cette instruction configure `batch` transformées de taille `nx` dont les types des éléments opérandes (éléments d'*entrée*) et résultat (éléments de *sortie*) sont déterminés par `CUFFT_Z2Z`. `CUFFT_Z2Z` est une constante prédéfinie qui spécifie que :

- les données opérandes seront des nombres complexes double précision,
- les données résultat seront des nombres complexes double précision.

Après avoir configuré la transformée, il suffit de l'exécuter à l'aide de la fonction :

```
cufftExecZ2Z(plan, devPtr, devPtr, CUFFT_FORWARD);
```

Cette fonction exécute une FFT « sur place » (*in-place*) dont les variables d'entrée (opérandes) et de sortie (résultat) sont des nombres complexes double précision [22]. La constante `CUFFT_FORWARD` permet de préciser que la fonction exécute une transformée de Fourier directe. La variable `plan` donne accès à la configuration spécifiée par `cufftPlan1d`.

La lettre `Z` utilisée dans la dénomination de `CUFFT_Z2Z` et de `cufftExecZ2Z` permet de signaler que les données manipulées sont des nombres complexes double précision.

Une fois la transformée exécutée, il ne reste plus qu'à transférer le résultat sur le CPU, convertir chaque nombre de type `cufftDoubleComplex` en deux nombres :

- l'un représentant la partie réelle,
- l'autre représentant la partie imaginaire,

pour que le nombre complexe puisse être renvoyés à MATLAB.

On transfère le résultat vers le CPU avec l'instruction habituelle :

```
cudaMemcpy(h_B, devPtr, sizeof(cufftDoubleComplex)*batch*nx, cudaMemcpyDeviceToHost);
```

Avant de convertir chaque donnée complexe en deux valeurs séparées, on initialise le pointeur `plhs[0]` sur la zone mémoire qui recevra la matrice résultat. Pour cela, on utilise l'instruction :

```
plhs[0] = mxCreateDoubleMatrix(nx,batch, mxCOMPLEX);
```

Cette instruction crée donc un tableau prévu pour contenir une matrice de `nx` lignes et `batch` colonnes. `mxCOMPLEX` sert à signaler que les données à mettre dans le tableau `mxArray` comportent une partie imaginaire.

Maintenant que le pointeur `plhs[0]` est initialisé, on peut l'utiliser directement pour stocker chaque partie réelle et chaque partie imaginaire. Pour séparer partie réelle et partie imaginaire de chaque nombre, il n'existe pas de fonction CUFFT prédéfinie. Par conséquent, on crée la fonction suivante très similaire à la fonction `pack_dc2dc()` créée précédemment :

```
void unpack_dc2dc(cufftDoubleComplex *input, double *output_re, double *output_im, int Ntot)
 {
 int n;
 for(n=0; n<Ntot; n++)
        {
        output_re[n] = input[n].x;
        output_im[n] = input[n].y;
        }
 }
```

On fait appel à cette fonction grâce à l'instruction :

```
unpack_dc2dc(h_B,mxGetPr(plhs[0]),mxGetPi(plhs[0]),nx*batch);
```

avec :

- `mxGetPr(plhs[0])` : adresse de la partie réelle des données à renvoyer à MATLAB,
- `mxGetPi(plhs[0])` : adresse de la partie imaginaire des données à renvoyer à MATLAB.

Une fois les données renvoyées à MATLAB par l'intermédiaire de `plhs`, il ne reste plus qu'à libérer les ressources allouées en utilisant l'instruction : `cufftDestroy(plan);`

Puis on libère la mémoire GPU et CPU avec les fonctions `cudaFree` et `mxFree`.

## IV.4 Analyse des résultats pour l'API CUFFT

Comme pour l'API CUBLAS, ce n'est pas l'utilisateur qui fixe la taille des blocs et le nombre de registres. C'est l'API CUFFT qui fixe le nombre de tâches par bloc et le nombre de registres par tâche au moment de la compilation. C'est pourquoi dans un programme CUFFT, l'instruction ***-maxrregcount*** n'a aucune influence sur les résultats et la taille des blocs n'est pas configurable par l'utilisateur.

Le compilateur *nvcc* (voir §II.1) détermine ces paramètres à partir de la taille de la matrice résultat uniquement.

La transformée de fourier a été réalisée sur nombres flottants double précision avec partie imaginaire avec l'option ***-arch=sm_20*** dans la ligne de commande du compilateur (voir §II.5).

La communication des données entre MATLAB et le programme CUDA se fait via une fonction MEX. L'exactitude des données a été vérifiée par MATLAB avec la même méthode que pour le programme CUDA (voir §II.5).

Les tests (que ce soit par MATLAB ou par le profiler) ont tous été réalisés en mode Release. Les essais en mode Debug ne sont pas mentionnés.

Comme pour CUDA, les résultats (que ce soit pour MATLAB seul ou pour la fonction MEX-CUDA-C combinée à MATLAB) ont été moyennées sur 10 valeurs avec une exécution préalable non prise en compte car due aux incertitudes des mesures de temps lors de la mise en place des ressources de la carte graphique.

Comme pour CUBLAS et contrairement au programme CUDA, un programme utilisant l'API CUFFT ne nécessite pas d'avoir des tailles de matrice résultat multiples de la taille des blocs. La taille de transformée peut donc être très variable. Comme pour CUBLAS, le compilateur choisit lui-même les paramètres tels que :

- le nombre de registres à utiliser par tâche,
- la taille des blocs,
- le nombre de blocs résidents.

### IV.4.1 Calcul du nombre de FLOPS

La façon de calculer le nombre de FLOPS dépend de l'algorithme de calcul utilisé :

- MATLAB utilise l'algorithme de la FFTW,
- L'API CUFFT utilise un algorithme basé sur la FFTW mais elle utilise surtout l'algorithme de Cooley-Tukey.

#### IV.4.1.1 Cas de MATLAB

La FFTW est l'implémentation la plus rapide de l'algorithme de la FFT. Elle peut calculer des transformées de n'importe quelle taille sur données réelles ou complexes. L'efficacité de la FFTW tient dans le fait qu'elle utilise plusieurs algorithmes et elle choisit le plus adapté selon les circonstances. Pour décomposer des trasnformées de taille :

- composite (multiple d'un nombre premier) en transformées plus petites, la FFTW choisit parmi plusieurs variantes de l'algorithme de Cooley-Tukey.
- égale à un nombre premier, elle utilise soit l'algorithme de Rader soit l'algorithme de Bluestein.

Son fonctionnement est optimal pour des tableau de tailles multiples de petits nombres premiers (*2*, *3*, *5* ou *7*), une taille égale à un nombre premier élevé étant le pire cas mais exécutant malgré tout ($n*log_2\ n$) opérations.

Etant donné que l'algorithme le plus défavorable parvient à exécuter ($n*log_2\ n$), le nombre de FLOPS réalisés par chaque exécution de MATLAB seul a donc été calculé de la manière suivante : $FLOPS = \frac{batch * (nx * \log_2 nx)}{tpscalcul}$.

### IV.4.1.2 Cas de l'API CUFFT

L'API CUFFT [22] utilise l'algorithme de Cooley-Tukey pour réduire le nombre d'opérations nécessaires, et pour ainsi optimiser la performance selon la taille de transformée. C'est pourquoi la performance est la meilleure lorsque la taille de la transformée peut être factorisée en $2^a * 3^b * 5^c * 7^d$ avec *a*, *b*, *c* et $d \geq 0$. Pour les autres tailles, les transformées à une seule dimension sont prises en charge par l'algorithme de Bluestein, lui-même construit à partir de l'algorithme de Cooley-Tukey.

La précision d'une implémentation Bluestein diminue lorsque la taille augmente, surtout en mode simple précision, en raison de l'accumulation d'imprécisions des opérations en virgule flottante.

Cependant, dans le cadre de ce mémoire, les FFT réalisées ayant toutes des tailles multiple de 2, la méthode choisie pour le calcul des FLOPS sera celle basée sur l'algorithme de Cooley-Tukey pour tous les tests.

Le nombre de FLOPS réalisés par chaque exécution de la fonction MEX-CUFFT-C a donc été calculé de la manière suivante : $FLOPS = \frac{batch * (nx * \log_2 nx)}{tpscalcul}$.

## IV.4.2 Influence de la taille d'une transformée sur la performance

Avec un nombre de transformées réduit à une seule transformée, on obtient les temps de calcul suivants représentés sur la Figure 46. Bien que ce ne soit pas visible sur la Figure 46, les courbes se croisent pour une taille de transformée de $2^{14}$ et un temps d'exécution de 0,8ms. Ce croisement est représenté sur la Figure 47.

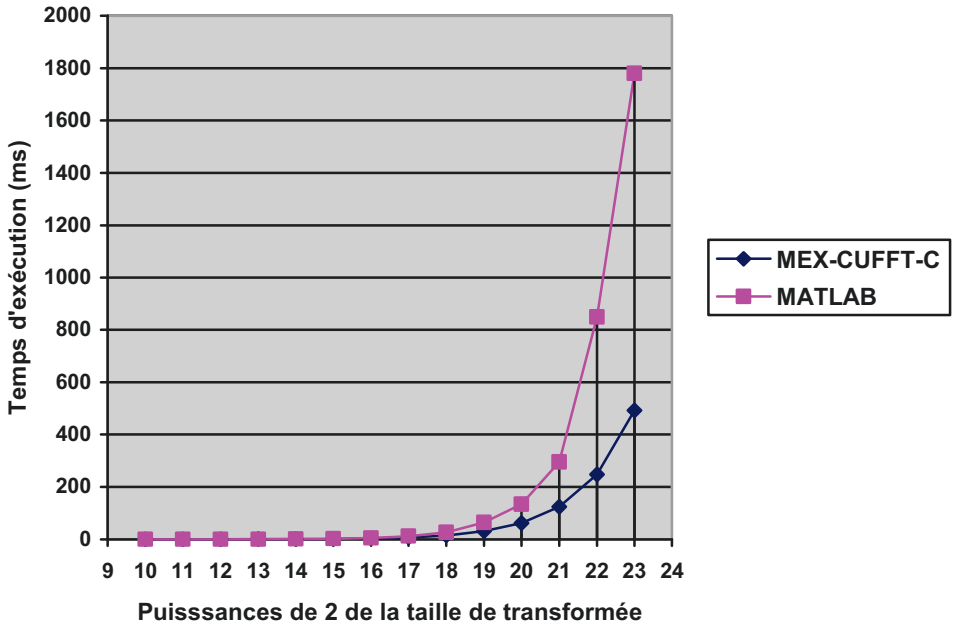



Figure 46 : Comparaison des temps d'exécution de la fonction MEX-CUFFT-C et de MATLAB seul

Attention : pour une taille de transformée de $2^{23}$ (Figure 47), la mémoire est parfois insuffisante. Les temps obtenus avec cette taille de transformée ne sont donc mentionné qu'à titre indicatif. La mémoire est toujours insuffisante pour un nombre de transformées supérieur à $2^{23}$.

Sur la Figure 47, on remarque que tant que la taille de transformée reste supérieure ou égale à $2^{15}$, la fonction MEX-CUFFT-C est plus performante que MATLAB.

On remarque que plus les tailles de transformée sont grandes, plus la fonction MEX-CUFFT-C est performante par rapport à MATLAB seul.

On peut donc considérer que :

- pour des petites tailles de FFT (taille $\leq 2^{14}$), il est préférable d'utiliser MATLAB seul,
- pour des grandes tailles de FFT (taille $> 2^{14}$), il est préférable d'utiliser la fonction MEX-CUFFT-C.

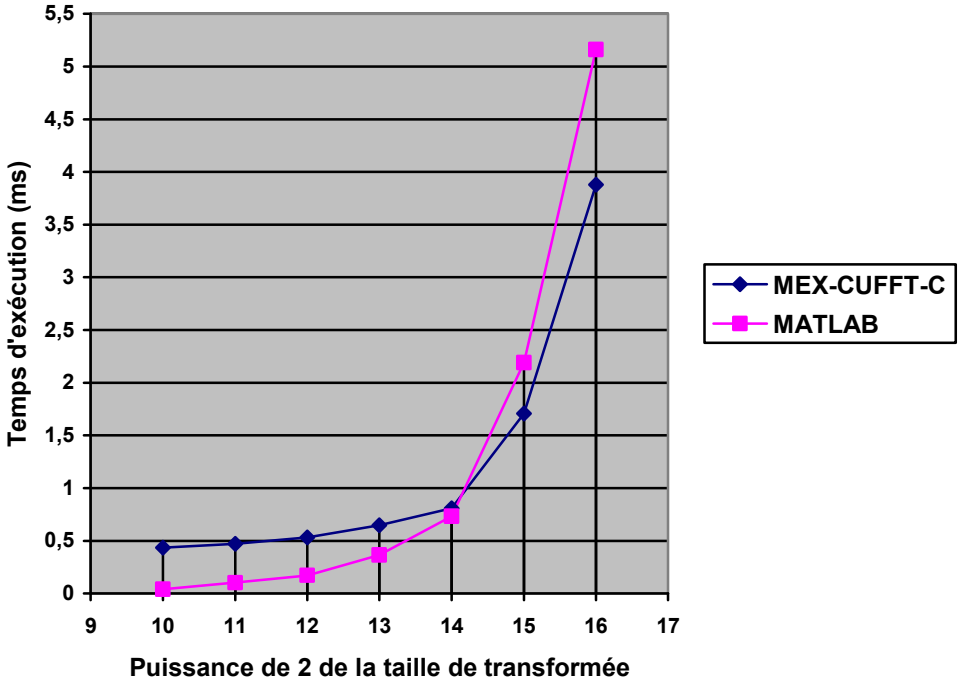



Figure 47 : Comparaison des temps d'exécution de la fonction MEX-CUFFT-C et de MATLAB seul

### IV.4.3 Influence du nombre de transformées de taille fixe sur la performance

Pour l'exécution d'un nombre variable de transformées de taille fixe ($2^{15}$), on obtient les temps de calcul représentés sur la Figure 48.

Attention, pour 256 transformées de taille égale à $2^{15}$, la mémoire est parfois insuffisante. Les temps obtenus avec cette taille de transformée ne sont donc mentionné qu'à titre indicatif. La mémoire est toujours insuffisante pour un nombre de transformées supérieur à 256.

On remarque sur la Figure 48 que le nombre de transformées n'a que peu d'influence sur la performance. A partir d'une taille de transformée de $2^{15}$, la fonction MEX-CUFFT-C est systématiquement plus performante que MATLAB seul quelque soit le nombre de transformées à exécuter (dans les limites de la mémoire).

On remarque que la performance de n transformées exécutées simultanément reste globalement peu différente de la performance de n transformées exécutée séquentiellement. On remarque que les temps d'exécution double lorsque le nombre de transformées double.

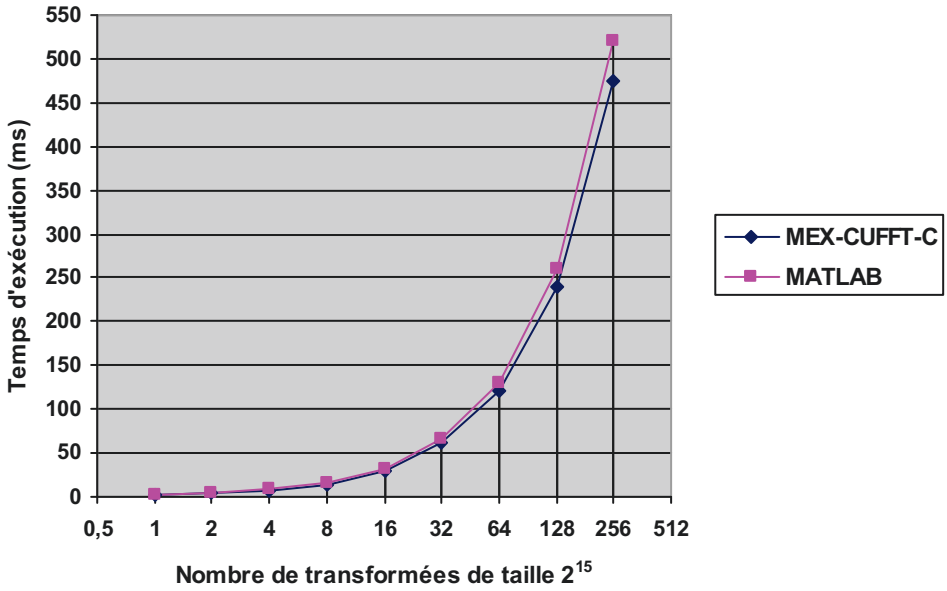



Figure 48 : Comparaison des temps d'exécution de la fonction MEX-CUFFT-C et de MATLAB selon le nombre de transformées de taille fixe ($2^{15}$) exécutées simultanément

### IV.4.4 Procédure de choix des paramètres par l'API CUFFT

La configuration adoptée (nombre de registres, taille des blocs et taille de grille) par l'API CUFFT dépend du type de taille de transformée (taille égale à un nombre premier ou taille multiple d'un nombre et lequel). Par conséquent, les tests ayant tous été réalisés avec des tailles de transformées multiples de 2, la configuration adoptée par l'API CUFFT a été la même dans tous les cas, c'est-à-dire :

- une taille de bloc de 256 lignes et une colonne,
- 22 registres par tâche,

résultant ainsi en :

- 5 blocs résidents par multiprocesseur,
- un taux d'occupation de 83,3%.

Comme pour CUBLAS,

- le nombre de blocs résidents que le gestionnaire de tâches attribue par multiprocesseur,
- le taux d'occupation,

sont déductibles des mêmes conditions que précédemment (voir §II.4.2.7).

Le nombre de blocs résidents est fixé à 5 par le gestionnaire de tâches, ce qui permet de satisfaire :

- la condition 1 : 22 registres/tâche * 256 tâches/bloc * 5 blocs = 28160 < 32768,
- la condition 2 : 256 tâches/bloc * 5 blocs résidents = 1280 tâches < 1536,
- la condition 3 : 0 ko utilisé < 48 ko car d'après le profiler, la CUFFT n'utilise jamais la mémoire partagée,
- la condition 4 : 5 blocs résidents < 8 blocs résidents maximum.

Cette configuration adoptée par le gestionnaire de tâches entraîne un taux d'occupation de 83,3%.

# V Synthèse des résultats CUDA, CUBLAS et CUFFT

D'après l'étude d'un programme MEX-CUDA-C et d'un programme MEX-CUBLAS-C, on a constaté de façon claire la performance supérieure de CUDA et CUBLAS sur MATLAB utilisé seul pour les matrices résutat de grande taille (supérieure ou égale à *960*960* pour CUDA et supérieure ou égale à *400*400* pour CUBLAS). La programmation CUDA est plus souple que la programmation CUBLAS dans la mesure où on peut contrôler la taille des blocs.

Cependant la taille de matrice résultat doit être multiple de la taille de bloc, ce qui restreint les possibilités alors que la programmation CUBLAS accepte n'importe quelle taille de matrice résultat puisque les paramètres sont ensuite choisis par le compilateur en fonction de cette taille de matrice résultat. De plus la performance de CUBLAS est nettement meilleure que la performance de CUDA.

D'autre part, on pourrait se demander à quel niveau se situe la différence de performance entre CUDA et CUBLAS. La différence de performance la plus marquée se situe-t-elle au niveau :

- transfert MATLAB-programme ?
- transfert CPU-GPU ?
- kernel ?

Pour le savoir il suffit de comparer le temps d'exécution total de la fonction MEX-CUDA-C avec les temps de transfert ou d'exécution de kernel affichés par le profiler. On fait le même raisonnement pour la fonction MEX-CUBLAS-C et on obtient le Tableau 14 et le Tableau 15 suivants.

Sur ces tableaux, les temps de transfert CPU → GPU sont divisés en deux parties par le signe « + ». Le 1$^{er}$ et le 2$^{ème}$ nombre représentent respectivement les temps de transfert de la 1$^{ère}$ matrice opérande et de la 2$^{ème}$ matrice opérande. A noter que le temps de transfert de la 1$^{ère}$ matrice est systématiquement plus long que le temps de transfert de la 2$^{ème}$ matrice. Cette différence de temps est encore plus marquée pour la fonction MEX-CUBLAS-C que pour la fonction MEX-CUDA-C.

Tableau 14 : Comparaison des durées d'exécution de la fonction MEX-CUDA-C totale et des durées d'exécution de certaines fonctions prises isolément.

| Taille de la matrice résultat | Fonction MEX-CUDA-C | | | | |
|---|---|---|---|---|---|
| | Temps d'exécution (ms) | Kernel seul (ms) | Temps de transfert CPU → GPU (ms) | Temps de transfert GPU → CPU (ms) | Temps total (kernel + transferts) (ms) |
| 1024*1024 | 71,7 | 21,7 | 3,64+3,53 | 4,46 | 33,3 |
| 960*960 | 66,8 | 17,9 | 3,34+3,15 | 4,02 | 28,4 |
| 896*896 | 62,0 | 14,5 | 2,90+2,80 | 3,50 | 23,7 |

Tableau 15 : Comparaison des temps d'exécution affichés par MATLAB et affichés par le profiler pour la fonction MEX-CUBLAS-C

| Taille de la matrice résultat | Fonction MEX-CUBLAS-C | | | | |
|---|---|---|---|---|---|
| | Temps d'exécution (ms) | Kernel seul (ms) | Temps de transfert CPU → GPU (ms) | Temps de transfert GPU → CPU (ms) | Temps total kernel et transferts |
| 1024*1024 | 36,7 | 17,0 | 4,06+3,51 | 3,08 | 27,7 |
| 960*960 | 31,5 | 14,0 | 4,41+3,25 | 3,11 | 24,8 |
| 896*896 | 26,1 | 11,3 | 3,72+2,93 | 2,76 | 20,7 |

Quand on observe les tableaux ci-dessus, on remarque que les temps de transfert sont très peu différents entre CUDA et CUBLAS (les temps de transfert CUBLAS sont même supérieurs en moyenne à ceux de CUDA). D'autre part, le temps d'exécution du kernel CUDA ne dépasse jamais de plus de 5,6 ms le temps d'exécution du kernel CUBLAS. Par contre, les temps totaux d'exécution CUDA sont approximativement deux fois plus important que ceux de CUBLAS.

Pour l'API CUFFT, on a pu voir qu'un programme de FFT utilisant cette bibliothèque est plus performant qu'un programme MATLAB exécutant la même fonction à condition d'avoir une taille de tranformée très grande (supérieure à 65536).

Après l'étude de CUDA et de CUBLAS, on peut dégager plusieurs points importants :

Les facteurs déterminants dans les performances CUDA sont :

1. Les facteurs indispensables au fonctionnement correct d'un programme :
   - Le nombre de tâches par bloc doit être un multiple de 32 (nombre de tâches par warp).
   - La taille de chaque dimension de grille doit être multiple des dimensions de bloc respectives
2. Les facteurs utiles pour améliorer les performances :
   chaque multiprocesseur ne peut pas traiter :
   - un ensemble de blocs résidents :
     - nécessitant plus de registres que les 32768 registres qu'il possède.
     - comportant au total plus de 1536 tâches résidentes.
     - nécessitant plus que les 48 ko de mémoire partagée qu'il possède.
   - plus de 8 blocs résidents à la fois.

   Dans le cas où une ou plusieurs de ces conditions ne serait pas vérifiées, le gestionnaire de tâches du processeur graphique se chargerait de réduire automatiquement le nombre de blocs résidents jusqu'à ce que les 4 conditions ci-dessus soient réunies.

Les différences entre CUBLAS et CUDA sont les suivantes :

- CUDA est relativement flexible au niveau de sa programmation mais ses fonctions ne sont pas optimisées par rapport à CUBLAS.
- CUBLAS est peu flexible mais comporte des fonctions de haut niveau optimisées pour les calculs sur matrices et vecteurs, ce qui le rend plus performant que CUDA sur ce type de calculs.
- Malgré son aspect plus éloigné de la programmation C que ne l'est CUDA, CUBLAS est néanmoins plus rapide à prendre en main.

L'idéal serait d'utiliser :

- CUBLAS pour tous les calculs sur matrices ou sur vecteurs,
- CUDA sur des calculs non gérable par CUBLAS,
- MATLAB seul pour les calculs sur matrice résultat de petite taille (<400*400).

De plus, ce mémoire ayant été réalisé sur une carte graphique n'utilisant que 448 cœurs (14 multiprocesseurs) sur les 512 que comporte une architecture Fermi complète, il pourrait être intéressant d'utiliser une carte utilisant la totalité des 512 cœurs (16 multiprocesseurs) de l'architecture, par exemple la GTX 580.

# VI Application de CUDA™ à la mise en œuvre d'algorithmes de télécommunications

## VI.1 Présentation

Une application possible de CUDA serait le traitement de signaux radar dans le cadre :

- soit d'un calcul de distance,
- soit d'un calcul de vitesse.

Une manière de mesurer la distance à un objet est d'émettre un train de courtes impulsions de signal radio, et de mesurer le temps que prend chaque impulsion pour revenir après avoir été réfléchie. On peut également remplacer le train d'impulsions par un signal pseudo aléatoire, comme cela sera le cas dans ce chapitre. La vitesse de l'onde étant constante, on peut facilement en déduire la distance de l'objet. On appelle *écho* l'onde réfléchie par la cible.

La vitesse peut être mesurée par décalage doppler dans un radar à ondes continues [23] également appelé *radar à ondes entretenues*. La vitesse mesurée n'est que la composante radiale de la vitesse de l'objet, c'est à dire la partie du vecteur vitesse projetée sur l'axe radar-cible. Un objet se rapprochant « comprime » l'onde. De par la vitesse de la cible, l'écho a une fréquence plus élevée que l'onde reçue par la cible. Inversement, lorsque la cible s'éloigne du radar, l'onde réfléchie par la cible a une fréquence plus faible que la fréquence reçue par la cible.

Dans le cadre des signaux radar, dans l'expression *ondes continues*, le mot *continu* se rapporte au fait que le signal (sinusoïdal) est émis en permanence. L'onde continue peut être :

- soit à fréquence constante (pour la mesure de la vitesse seule),
- soit fréquence modulée, c'est-à-dire que la fréquence varie à intervalles réguliers (pour la mesure de la vitesse et de la distance) :
  - le décalage temporel permet de mesurer la distance,
  - le décalage (augmentation ou diminution) en fréquence permet de mesurer la vitesse d'un objet (qui se rapproche ou s'éloigne).

## VI.2 Utilisation d'un radar pseudo aléatoire

Dans le cadre de cette étude, prenons le cas d'un radar pseudo aléatoire pour une mesure de distance. Pour pouvoir comparer le signal pseudo-aléatoire émis et le signal reçu (écho), ce radar doit effectuer un calcul de corrélation.

Le fonctionnement global d'un tel radar est représenté par le schéma de la Figure 49 et par les étapes ci-dessous.

Le radar doit :

- stocker en mémoire une séquence pseudo aléatoire,
- émettre en boucle cette séquence pseudo-aléatoire vers l'objet dont on souhaite connaître la distance,
- comparer par corrélation l'écho $y(n)$ avec le signal pseudo aléatoire $x(n)$ stocké en mémoire,
- détecter le point pour lequel la corrélation est maximale.

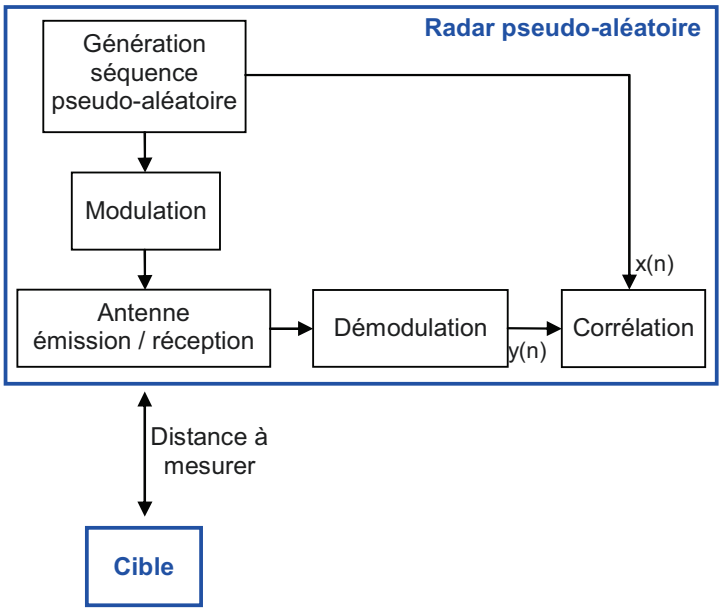



Figure 49 : Mesure de distance par radar

## VI.3 Lien entre transformée de Fourier et corrélation

Pour détecter un écho de radar, le meilleur moyen est de faire un produit de corrélation. On effectue le calcul de corrélation en utilisant les propriétés de la transformée de Fourier, ce qui limite le nombre d'opérations à effectuer.

Une de ces propriétés permet de réaliser l'opération :

$$TF(x(n) * y(n)) = X(k) \times Y(k)$$

Avec :
$x(n) * y(n)$ : produit de convolution de *x(n)* et *y(n)*
*x(n)* et *y(n)* : signaux temporels discrets
*X(k)* et *Y(k)* : transformées de Fourier respectives de *x(n)* et *y(n)*

On sait également que :

- Produit de convolution : $x(k) * y(k) = \sum_{n=0}^{N-1} x(n) \times y(k-n)$
- Produit de corrélation : $\Gamma_{xy}(k) = \sum_{n=0}^{N-1} \overline{x}(n) \times y(n+k) = \sum_{n=0}^{N-1} \overline{x}(n-k) \times y(n)$

En supposant dans les deux cas le signal reçu *y(n)* en retard par rapport au signal émis *x(n)*.

Il suffit ensuite de remarquer qu'entre le produit de convolution et le produit de corrélation, il n'y a qu'un signe de différence. On en déduit donc que : $TF(\Gamma_{xy}(k)) = \overline{X}(k) \times Y(k)$

Pour trouver le produit de corrélation, il suffit donc de faire :

- la TFD de *x(n)* suivie de son conjugué.
- la TFD de *y(n)*.
- la multiplication des deux TFD.
- la TFD inverse du produit des deux TFD.

## VI.4 Application de CUDA™ à la détection d'un objet

Pour faire le calcul d'une TFD sous CUDA, on peut décomposer le calcul en se basant sur un algorithme de FFT. Cependant, l'environnement CUDA contient une API optimisée pour le calcul de FFT d'une complexité en $N*log_2N$ : l'API CUFFT. Le calcul pourra donc se faire par cette bibliothèque de fonctions.

L'ensemble du calcul peut donc être résumé par l'algorithme de la Figure 50.

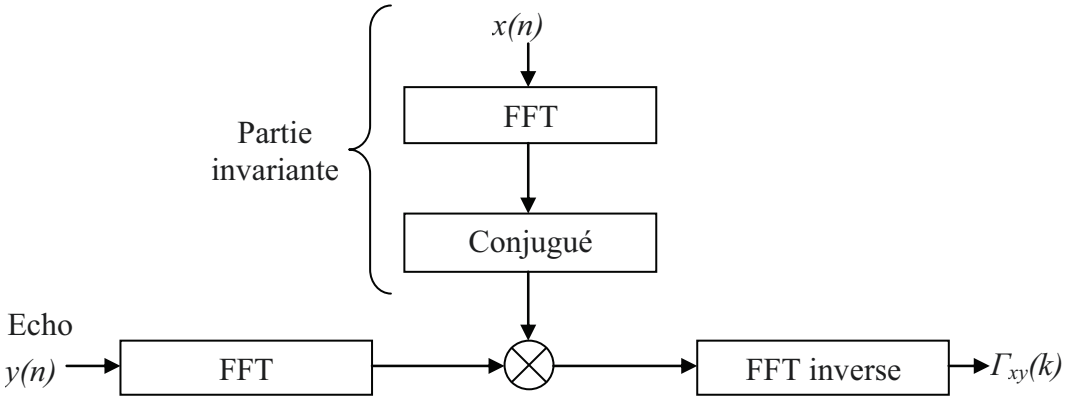



Figure 50 : Algorithme de corrélation rapide

On veut connaître le temps que nécessite une corrélation. La partie « temps réel » de la corrélation comprend :

- une FFT directe,
- un produit scalaire,
- une FFT inverse.

On rappelle les temps d'exécution d'une FFT directe avec CUFFT dans le Tableau 16.

Tableau 16 : Temps d'exécution d'une FFT avec CUFFT

| Taille de transformée | Temps d'exécution (ms) |
|---|---|
| $2^{19}$ | 31,3 |
| $2^{20}$ | 62,3 |
| $2^{21}$ | 124 |
| $2^{22}$ | 248 |

On souhaite estimer le temps d'exécution d'un produit scalaire complexe sous CUBLAS. On rappelle que la multiplication de matrices avec CUBLAS a atteint 58,51 GFLOPS. On se basera donc pour cette estimation de temps sur une durée d'exécution par opération de $\frac{1}{58,51\times10^9}$ secondes.

La multiplication de deux nombres complexes nécessite 4 multiplications et 3 additions. Dans le cas d'une corrélation de taille $2^{22}$, Le produit scalaire nécessite donc :

- $4*2^{22}$ multiplications,
- $3*2^{22} - 1 \approx 3*2^{22}$ additions,

c'est-à-dire $7*2^{22}$ opérations. On peut évaluer la durée approximative $d$ qu'aurait eu le produit scalaire sous CUBLAS : $d = \frac{7*2^{22}}{58,51\times10^9} = 0,502ms$

Dans le cas d'une taille de transformée de $2^{22}$, le temps d'exécution total de la corrélation est donc de 2*248 + 0,502=496,5 ms.

Pour les autres tailles de transformée, on obtient les valeurs du Tableau 17.

Tableau 17 : Temps d'exécution total de la corrélation

| Taille de transformée | Temps d'exécution (ms) | Temps d'exécution de la corrélation (ms) |
|---|---|---|
| $2^{19}$ | 31,3 | 63,1 |
| $2^{20}$ | 62,3 | 125,1 |
| $2^{21}$ | 124 | 248,5 |
| $2^{22}$ | 248 | 496,5 |

Dans le cas de la corrélation entre un signal envoyé et son écho, le calcul de la corrélation par CUDA (CUFFT) se fera en deux étapes principales (préliminaire et secondaire) comprenant plusieurs étapes intermédiaires.

Etape préliminaire :

- calcul sur CPU de la FFT d'un signal pseudo-aléatoire *x(n)*.
- calcul sur CPU du conjugué de la FFT de *x(n)*.
- transfert CPU vers mémoire globale GPU du conjugué de la FFT de *x(n)*.

Etapes principales des calculs réalisés en permanence (temps réel) :

- envoi du signal d'émission par l'antenne.
- échantillonnage permanent des signaux d'entrée et stockage des échantillons (sur le CPU).
- prélèvement d'une « fenêtre » d'échantillons *y(n)* parmi les échantillons stockés en mémoire et transfert de cette fenêtre du CPU vers la mémoire globale $GPU_{réception}$.
- calcul (avec l'API CUDA) de la FFT de *y(n)* .
- calcul GPU (avec l'API CUDA) du produit de $\overline{X}(k)$ par *Y(k)* avec utilisation de la mémoire partagée.
- calcul GPU (avec l'API CUDA) de la FFT inverse du résultat du produit de $\overline{X}(k)$ par *Y(k)*.
- renvoi du résultat de la corrélation (résultat de la FFT inverse) vers le CPU.

## VI.5 Evaluation des performances de CUDA par rapport à un radar réel

Idéalement, le temps de calcul doit rester inférieur à la période de récurrence de la séquence pseudo-aléatoire. Le nombre d'éléments du signal pseudo-aléatoire définit le gain de traitement. S'il y a $2^n$ éléments, le gain de traitement sera de 3n dB.

D'après le Tableau 17, si à 62 ms de temps de récurrence, on peut monter à $2^{19}$ éléments avec une carte graphique, on peut monter à $2^{18}$ seulement avec MATLAB. On gagne donc 3dB de gain de traitement en passant d'un calcul par MATLAB à un calcul par CUFFT.

Cependant, on doit tenir compte du fait que le temps de récurrence est lié aux signaux à observer. En particulier, la plage Doppler observable est liée au temps de récurrence par la relation :

$$Doppler = \frac{1}{2 * Tr}$$ où *Tr* représente le temps de récurrence.

Par conséquent, l'application de CUDA au calcul de distance par radar ne fonctionne qu'à condition de considérer une application pour laquelle la plage Doppler serait faible. Par exemple, un sondeur ionosphérique pourrait convenir étant donné que cet instrument est basé sur ce principe.

Cependant, on doit se rendre compte qu'on reste dans des limites faibles, même pour un sondeur ionosphérique. L'intérêt serait de paralléliser plusieurs FFT plus petites. Ceci peut être intéressant dans la mesure où on obtient un gain de temps.

# Conclusion

Ce mémoire avait pour objectif de réaliser l'étude et le test de CUDA™ (NVIDIA) appliqué à la mise en œuvre d'algorithmes de télécommunications.

Une partie de l'objectif était donc d'étudier dans quelle mesure, le processeur d'une carte graphique NVIDIA pourrait éventuellement remplacer le CPU pour des calculs de grandes dimensions, voire le devancer au niveau de la performance.

Dans un premier temps, une présentation générale des cartes graphiques NVIDIA a été faite. Un historique des cartes graphiques NVIDIA a donc été établi suivi d'un bref récapitulatif des principales parties d'une carte graphique, puis un descriptif de l'évolution des architectures GPU des cartes NVIDIA.

Ainsi, nous avons pu observer de quelle manière les cartes graphiques NVIDIA sont passées d'un ensemble de pipelines de traitement graphique de plus en plus complexes à des processeurs de traitement graphique. Puis, nous avons vu comment ces processeurs de traitement ont progressivement été améliorés en passant d'un traitement graphique à un traitement à finalité non graphique.

Il s'est alors avéré que la carte graphique présente au départ dans le PC dédié au projet (carte NVIDIA 8400GS comportant 8 cœurs de traitement) était particulièrement peu performante par rapport aux dernières générations de carte graphique. Nous avons donc procédé à son remplacement par une des dernières cartes graphiques sorties sur le marché au moment de ce stage : la GTX 470 (comportant 448 cœurs de traitement).

Puis nous nous sommes lancés dans l'étude approfondie de CUDA™. Ainsi nous avons vu dans un premier temps un programme simple utilisant la bibliothèque CUDA, puis nous avons étudié l'intégration d'un programme CUDA dans une foncton MEX pour permettre la communication de données avec MATLAB. Et enfin, nous avons étudié un programme de multiplication de matrices utilisant les ressources de la carte graphique de manière optimale. Ces ressources étant limitées, nous avons vu la manière de gérer la mémoire partagée et les transferts de données de manière optimale.

Puis, suite à des performance de CUDA relativement décevantes (en 2010) par rapport à ce qu'il en est dit par de nombreux experts, il a été découvert sur internet que l'API CUBLAS (bibliothèque CUDA dédiée à l'exécution de calculs optimisés sur matrices ou vecteurs) pouvait améliorer la performance de manière significative. Cette bibliothèque a donc été étudiée et s'est effectivement avérée plus performante pour la multiplication de matrices.

Nous avons pu voir que la bibliothèque CUBLAS semble produire de bonnes performances bien qu'on aurait pu s'attendre à de bien meilleures performances. De plus, le temps de prise en main et de mise en œuvre de CUBLAS est beaucoup plus court que pour CUDA seul.

Puis, ce mémoire ayant également pour but d'être appliqué à la mise en œuvre d'algorithmes de télécommunications, la bibliothèque CUFFT (bibliothèque CUDA consacrée aux FFT) a été également étudiée.

Suite au test de ces différentes bibliothèques, il s'est avéré que les performances de CUDA sont assez décevantes d'une manière générale. Le gain en temps d'exécution

par rapport à MATLAB seul est relativement faible pour une complexité de programmation assez importante.

Puis nous avons pu étudier une possibilité d'application de CUDA au traitement de signaux radar.

La fin de la rédaction de ce mémoire ayant eu lieu en 2013 alors que le stage s'est terminé en 2010, il est donc à noter que d'importantes évolutions ont eu lieu dans l'architecture des cartes graphiques. Dans ce mémoire, l'architecture Fermi (comprenant 14 multiprocesseurs de 32 cœurs chacun) a été étudiée en détails. En 2012 est apparue l'architecture Kepler qui comprend 15 multiprocesseurs, chacun composés de 192 cœurs. Ce mémoire ne prétend donc pas montrer les performances des dernières cartes graphiques mais les performances d'une des cartes graphiques les plus rapides du début de l'année 2010.

D'un point de vue personnel, malgré des performances de CUDA moins spectaculaires que ce que j'espérais, je suis plutôt satisfait d'avoir travaillé dans ce domaine que je connaissais très peu au départ. Je suis également satisfait d'avoir pu mettre en pratique mes connaissances de traitement du signal pour ce mémoire.

De plus, l'autonomie complète dont j'ai bénéficié sur ce mémoire m'a permis d'acquérir des compétences transférables à d'autres domaines : la communication de données entre MATLAB et un programme C, la programmation en langage C sur un projet concret, la découverte d'un langage de programmation autre que le C, la configuration et l'utilisation de logiciels de développement (MATLAB, Visual Studio et outils spécifiques CUDA), la recherche sur internet et enfin, la persévérance.

# Glossaire

| Terme | Description en anglais | Description ou equivalent en français |
|---|---|---|
| AGP | Accelerated Graphics Port | Port Graphique Accéléré |
| ALU | Arithmetic Logic Unit | Unité Arithmétique et Logique |
| AMD | Advanced Micro Devices | Nom d'un fabricant de semi-conducteurs |
| ANSI | American National Standards Institute | Institut national américain des normes |
| API | Application Program Interface | Interface de programmation |
| ATI | Array Technologies Incorporated | Nom d'une entreprise canadienne de production de processeurs graphiques |
| BIOS | Basic Input/Output System | Système élémentaire d'entrées/sorties |
| BLAS | Basic Linear Algebra Subprograms | Bibliothèque de fonctons spécialisée dans les opérations de base de l'algèbre linéaire |
| CPU | Central Processing Unit | Unité centrale |
| CUBLAS | CUDA Basic Linear Algebra Subprograms | API CUDA spécialisée dans les Sous-programmes d'Algèbre Linéaire de Base |
| CUDA | Computer Unified Device Architecture | Architecture de calcul unifiée |
| CUFFT | CUDA Fast Fourier Transfom | API CUDA spécialisée dans l'exécution de Transformée de Fourier Rapide |
| CURAND | CUDA Random | API CUDA spécialisée dans la génération de nombres aléatoires CUDA |
| CUSP | CUDA Sparse | API CUDA spécialisée dans les calculs sur matrices creuses CUDA (Projet Google) |
| CUSPARSE | CUDA Sparse | API CUDA spécialisée dans les calculs sur matrices creuses CUDA |
| DDR | Double Data Rate | Débit de données double (sur front montant et descendant) |
| DLL | Dynamic Link Library | Bibliothèque de liens dynamiques |
| DRAM | Dynamic Random Access Memory | Mémoire dynamique à accès aléatoire |
| DVI-A | Digital Visual Interface Analog | Interface vidéo numérique pour les signaux analogiques |
| DVI-D | Digital Visual Interface Digital | Interface vidéo numérique pour les signaux numériques |
| DVI-I | Digital Visual Interface Integrated | Interface vidéo numérique pour les signaux analogique ou numérique |
| ECC | Error-Correcting Code | Code correcteur d'erreur |
| EISA | Extended Industry Standard Architecture | Norme d'amélioration du bus ISA |
| FFT | Fast Fourier Transform | Transformée de Fourier rapide |
| FFTW | Fastest Fourier Transform in the World | Transformée de Fourier la plus rapide du monde |
| FLOPS | Floating Point Operations per Second | Nombre d'opérations à virgule flottante exécutées par seconde |
| FMA | Fused Multiply-Add | Opération de multiplication et accumulation (résultat non arrondi pour la multiplication) qui met en œuvre la norme IEEE 754-2008 pour la virgule flottante |
| GDDR | Graphics Double Data Rate | Type de mémoire DDR spécialement conçue pour les cartes graphiques |
| GPGPU | General-Purpose Computing on Graphics Processing Units | Calcul générique sur processeur graphique |
| GPU | Graphics Processing Unit | Processeur Graphique |

| HD DVD | High Density Digital Versatile Disc | Disque numérique polyvalent de haute densité |
|---|---|---|
| HDMI | High Definition Multimedia Interface | Interface Multimédia Haute Définition |
| HPC | High Performance Computing | Calcul haute performance |
| IDE | Integrated Development Environment | Environnement de Développement Intégré |
| IEEE | Institute of Electrical and Electronics Engineers | Institut des ingénieurs électriciens et électroniciens |
| IRM | | Imagerie par Résonance Magnétique |
| ISA | Industry Standard Architecture | Norme de bus informatique |
| MAD | Multiply-Add | Opération de multiplication et accumulation (résultat arrondi pour la multiplication) qui met en œuvre la norme IEEE 754-1985 pour la virgule flottante |
| MCA | Micro Channel Architecture | Type de bus |
| MEX | Matlab Executable | Type de fonction externe à MATLAB mais exécutable sous MATLAB |
| MPEG | Moving Picture Experts Group | Groupe de travail ayant introduit plusieurs normes de traitement audio et vidéo |
| nlhs | Number of left-hand side arguments | Nombre d'arguments de sortie d'une fonction MEX |
| nrhs | Number of right-hand side arguments | Nombre d'arguments d'entrée d'une fonction MEX |
| NTSC | National Television System Committee | Norme de codage analogique de la vidéo en couleur |
| NuBus | NuMachine Bus | Nom d'un bus informatique parallèle initialement développé pour le projet NuMachine |
| NV43 | NVIDIA 43 | Nom de code d'un des premiers processeurs graphiques NVIDIA |
| NVCC | NVIDIA CUDA Compiler | Nom du compilateur CUDA |
| OpenGL | Open Graphics Library | API libre pour la conception 2D et 3D |
| PAL | Phase Alternating Line | Norme vidéo basée sur l'alternance de phase |
| PCI | Peripheral Component Interconnect | Nom d'une norme de bus interne permettant de connecter des cartes d'extension sur une carte mère |
| PCI-Express | Peripheral Component Interconnect - Express | Nom d'une norme de bus interne qui remplace les normes AGP et PCI |
| PCI-X | Peripheral Component Interconnect – X | Nom d'une évolution du bus PCI standard. Rapidement remplacé par le PCI-Express |
| PDB | Program Data Base | Base de données de programme |
| plhs | Pointer to left-hand side arguments | Pointeur sur les arguments de sortie d'une fonction MEX |
| prhs | Pointer to right-hand side arguments | Pointeur sur les arguments d'entrée d'une fonction MEX |
| PTX | Parallel Thread Execution | Machine virtuelle d'exécution de tâches parallèles. Type d'instructions générées par les langages CUDA ou C |
| RAM | Random Access Memory | Mémoire vive d'un ordinateur |
| RAMDAC | Random Access Memory Digital-to-Analog Converter | Convertisseur qui transforme l'image numérique en signal analogique |
| ROP | Raster Operations | Opérations de traitement d'image précédant le stockage de l'image en mémoire vidéo |
| SDK | Software Development Kit | Kit de développement logiciel |
| SECAM | | Séquentiel Couleur A Mémoire (système français de codage vidéo) |
| SFU | Special Function Unit | Unité de l'architecture GT200 et GF100 spécialisée dans l'exécution de fonctions particulières (sin, cos, racine carrée...) |

| | | |
|---|---|---|
| SIMD | Single Instruction Multiple Data | Instruction unique sur données multiples |
| SIMT | Single-Instruction Multiple-Thread | Instruction unique sur tâches multiples |
| SM | Streaming Multiprocessor | Multiprocesseur |
| SP | Streaming Processor | Processeur de flux ou coeur |
| SPA | Scalable Processor Array | Tableau de processeurs modulable (architectures G80). Contient un nombre modulable de TPC |
| SQL | Structured Query Language | Langage de programmation pour bases de données |
| ST | Store | Unité de stockage d'un multiprocesseur |
| STL | Standard Template Library | Nom d'une bibliothèque C++ |
| TPC | Texture Processing Cluster | Bloc de traitement de texture contenant un nombre modulable de multiprocesseurs |
| TPC | Thread Processing Cluster | Bloc de traitement de tâches contenant un nombre modulable de multiprocesseurs |
| UPA | Ultra Port Architecture | Architecture de bus d'interconnexion |
| USB | Universal Serial Bus | Norme de transmission série pour bus informatique |
| VESA | Video Electronics Standards Association | Groupe informatique définissant certaines normes vidéo |
| VGA | Video Graphics Array | Norme d'affichage pour ordinateurs |
| VLB | VESA Local Bus | Bus interne pour compatibles PC |

# Bibliographie


[1] KIRK David B., HWU Wen-mei W.. Programming massively parallel processors – A hands-on approach. Elsevier Inc.. Burlington, Massachusetts, 2010, 258 p.

[2] NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. Nvidia Corp. Santa Clara, Californie, 2006, 55 p.

[3] NVIDIA. COMPATIBILITY GUIDE FOR CUDA APPLICATIONS. Nvidia Corp., 2010, 12 p.

[4] NVIDIA. TUNING CUDA APPLICATIONS FOR FERMI – Application Note. Nvidia Corp., 2010, 11 p.

[5] NVIDIA. NVIDIA CUDA™ C Programming Guide; Version 3.2. Nvidia Corp. Santa Clara, Californie, 2010, 183 p.

[6] NVIDIA. NVIDIA CUDA™ C Best practices Guide; Version 3.2. Nvidia Corp. Santa Clara, Californie, 2010, 73 p.

[7] NVIDIA. Whitepaper NVIDIA's next Generation CUDA™ Compute Architecture. Nvidia Corp. 2009, 21 p.

[8] NVIDIA. NVIDIA GeForce GTX 200 GPU Architectural Overview. Nvidia Corp. 2008, 23 p.

[9] MSDN. Input-Assembler Stage (Direct3D 10), [en ligne]. Disponible sur : http://msdn.microsoft.com/en-us/library/bb205116(v=VS.85).aspx

[10] MSDN. Stream-Output Stage (Direct3D 10), [en ligne]. Disponible sur : http://msdn.microsoft.com/en-us/library/bb205121(v=VS.85).aspx

[11] MSDN. Rasterizer Stage (Direct3D 10), [en ligne]. Disponible sur : http://msdn.microsoft.com/en-us/library/bb205125(v=VS.85).aspx

[12] MSDN. Shader Stages (Direct3D 10), [en ligne]. Disponible sur : http://msdn.microsoft.com/en-us/library/bb205146(v=VS.85).aspx

[13] NVIDIA. Whitepaper Nvidia GF100. Nvidia Corp. Santa Clara, Californie, 2010, 31 p.

[14] NVIDIA. NVIDIA CUDA™ C Reference Manual; Version 3.2. Nvidia Corp. Santa Clara, Californie, 2010, 360 p.

[15] The Mathworks. MATLAB® 7 External interfaces. The Mathworks. Natick, Massachusetts, 2009, 750 p.

[16] Mathworks. Product Support, MEX-files Guide, [en ligne]. Disponible sur : http://www.mathworks.com/support/tech-notes/1600/1605.html

[17] MSDN. Standard Types, [en ligne]. Disponible sur : http://msdn.microsoft.com/en-us/library/323b6b3k.aspx

[18] NVIDIA. CUDA Technical Training, Volume I: Introduction to CUDA Programming. Nvidia Corp. Santa Clara, Californie, 2008, 94 p.

[19] NVIDIA. The CUDA Compiler Driver NVCC. Nvidia Corp. Santa Clara, Californie, 2010, 39 p.

[20] NVIDIA. CUBLAS Library. Nvidia Corp. Santa Clara, Californie, 2010, 256 p.

[21] National Science Foundation. BLAS (Basic Linear Algebra Subprograms), [en ligne]. Disponible sur : http://www.netlib.org/blas/

[22] NVIDIA. CUDA CUFFT LIBRARY. Nvidia Corp. Santa Clara, Californie, 2005-2011, 31 p.

[23] THOUREL L. Initiation aux techniques modernes des radars. Cepadues Editions. Toulouse, France, 1982, 304 p.

[24] Jean-Marie COLIN. Le radar – Théorie et pratique. Ellipses. Paris, France, 2002, 151 p.

# Table des annexes

# Annexe 1
# Comparaison et historique des cartes graphiques

Tableau 18 : Historique et comparaison des cartes graphiques des principaux constructeurs

| Année | ATI | NVIDIA | Matrox | Videologic et ou STMicroelectronics |
|---|---|---|---|---|
| 1995 | | STG2000 | Millenium | |
| 1996 | All-In-Wonder, Rage3D, Rage/Pro | | Mystique | PowerVR ou PCX1 |
| 1997 | 3D Rage Pro | Riva 128 | | |
| 1998 | Rage 128 GL, Rage 128 VR | TNT | G200 | |
| 1999 | Rage 128 Pro GL, Rage 128 Maxx | Vanta, TNT2, GeForce | G400 | Kyro |
| 2000 | Rage Fury Maxx, Radeon SDR, Radeon DDR, Radeon VE | GeForce 2 | G450 | Kyro 2 |
| 2001 | FireGL, Radeon 7000, 7200, 7500, 8500 | GeForce 3 | G550 | |
| 2002 | Imageon, Radeon 9000, 9100, 9500, 9700 | GeForce 4Ti | Parhelia | |
| 2003 | Radeon 9200, 9600, 9800 | GeForce 5200, 5600, 5700, 5800, 5900 | | |
| 2004 | Radeon 9250, 9550, X300, X500, X600, X700, X800, X850 | GeForce 4300, 5500, 5750, 5950, 6200, 6500, 6600, 6800 | | |
| 2005 | Radeon X1300, X1600, X1800 | GeForce 7800 | | |
| 2006 | Radeon X1600, X1650, X1900, X1950 | GeForce 7100, 7200, 7300, 7500, 7600, 7700, 7900, 7950, 7950GX2, 8800 | | |
| 2007 | Radeon HD 2400, HD 2600, HD 2900, HD 3800 | GeForce 8800, 8600, 8500, 8400, 8300 | | |
| 2008 | Radeon HD3870X2, HD4550, HD4650 HD4670, HD4850, HD4870, HD4850X2, HD4870X2 | GeForce 9200, 9300, 9400, 9500, 9600, 9800, GTX260, GTX280, 9800GX2, 9800 GTX | | |
| 2009 | Radeon HD 4770, HD4890, HD 5750, HD 5770, HD 5850, HD 5870, HD 5970 | GeForce GTX260+, GTX275, GTX295, GTX 285, G210, GT220, GT240 | | |
| 2010 | Radeon HD5450, HD5570, HD5670, HD5650, HD5750, HD5770, HD5850, HD5870, HD6850, HD6870, HD6950, HD6970 | GeForce GTX 480, GTX 470, GTX465, GTX 460, GTS 450, GT 430, GTX 580, GTX 570 | | |
| 2011 | Radeon HD6990 | GeForce GTX 560ti, GTX550ti, GTX590 | | |

Tableau 19 : Nombre de cœurs et de multiprocesseurs selon les cartes graphiques

| Cartes GeForce compatibles CUDA | Nombre de multiprocesseurs | Nombre de cœurs CUDA |
|---|---|---|
| GeForce 9300M GS, 9200M GS, 9100M G, 8400M G, G105M | 1 | 8 |
| GeForce GT 415M | 1 | 48 |
| GeForce 210, 310M, 305M | 2 | 16 |
| GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU, G210M, G110M | 2 | 16 |
| GeForce GT 435M, GT 425M, GT 420M | 2 | 96 |
| GeForce GT 445M | 3 | 144 |
| GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS | 4 | 32 |
| GeForce GTS 450, GTX 460M | 4 | 192 |
| GeForce GT 220, GT 330M, GT325M, GT 240M | 6 | 48 |
| GeForce 9700M GT, GT230M | 6 | 48 |
| GeForce GTX 470M | 6 | 288 |
| GeForce GTX 460 | 7 | 336 |
| GeForce 9600 GT, 8800M GTS, 9800M GTS | 8 | 64 |
| GeForce GT 335M | 9 | 72 |
| GeForce GTX 465, GTX 480M | 11 | 352 |
| GeForce 8800 GTS | 12 | 96 |
| GeForce GT 240, GTS 360M, GTS 350M | 12 | 96 |
| GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTS 260M, GTS 250M, 9800M GT | 12 | 96 |
| GeForce 9800 GT, 8800 GT, GTX 260M, 9800M GTX | 14 | 112 |
| GeForce GTX 470 | 14 | 448 |
| GeForce GTX 480 | 15 | 480 |
| GeForce GTX 580 | 16 | 512 |
| GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M | 16 | 128 |
| GeForce 8800 Ultra, 8800 GTX | 16 | 128 |
| GeForce 9800 GX2 | 2*16 | 2*128 |
| GeForce GTX 260 | 24 | 192 |
| GeForce GTX 285, GTX 280, GTX 275 | 30 | 240 |
| GeForce GTX 295 | 2*30 | 2*240 |

| Cartes Tesla compatibles CUDA | Nombre de multiprocesseurs | Nombre de cœurs CUDA |
|---|---|---|
| Tesla C2050 | 14 | 448 |
| Tesla C870 | 16 | 128 |
| Tesla D870 | 2*16 | 2*128 |
| Tesla S870 | 4*16 | 4*128 |
| Tesla C1060 | 30 | 240 |
| Tesla S1070 | 4*30 | 4*240 |

| Cartes Quadro compatibles CUDA | Nombre de multiprocesseurs | Nombre de cœurs CUDA |
|---|---|---|
| Quadro FX 380 LP, FX 380M, NVS 3100M, NVS 2100M | 2 | 16 |
| Quadro FX 370, NVS 290, NVS 160M, NVS 150M, NVS 140M, NVS 135M, FX 360M | 2 | 16 |
| Quadro 600 | 2 | 96 |
| Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M | 4 | 32 |
| Quadro 2000 | 4 | 192 |
| Quadro FX 880M, NVS 5100M | 6 | 48 |
| Quadro FX 2700M | 6 | 48 |
| Quadro FX 3600M | 8 | 64 |
| Quadro 4000 | 8 | 256 |
| Quadro FX 1800M | 9 | 72 |
| Quadro 5000M | 10 | 320 |
| Quadro 5000 | 11 | 352 |
| Quadro FX 2800M | 12 | 96 |
| Quadro FX 4600 | 12 | 96 |
| Quadro FX 3700 | 14 | 112 |
| Quadro 6000 | 14 | 448 |
| Quadro FX 4700 X2 | 2*14 | 2*112 |
| Quadro Plex 2100 D4 | 4*14 | 4*112 |
| Quadro FX 5600 | 16 | 128 |
| Quadro FX 3700M, FX 3800M | 16 | 128 |
| Quadro 1000 Model IV | 2*16 | 2*128 |
| Quadro Plex 2100 Model S4 | 4*16 | 4*128 |
| Quadro FX 4800 | 24 | 192 |
| Quadro FX 5800 | 30 | 240 |
| Quadro Plex 2200 D2 | 2*30 | 2*240 |

Tableau 20 : Principales caractéristiques de cartes graphiques

| Modèle (GeForce) | Nb de cœurs | Taille de mémoire (Mo) | GFLOPs | Puissance dissipée totale | Version de GPU (Compute capability) |
|---|---|---|---|---|---|
| 8400 GS | 8 | 512 | 33 | 25 | 1.1 |
| 9500 GT | 32 | 512 | 134,4 | 50 | 1.1 |
| GT 220 | 48 | 512 | 192 | 58 | 1.2 |
| 9600 GT | 64 | 512 | 312 | 95 | 1.1 |
| GT 240 | 96 | 512 | 385,9 | 69 | 1.2 |
| 9800 GT | 112 | 512 | 504 | 125/105 | 1.1 |
| GTS 250 | 128 | 512 | 705,024 | 145 | 1.1 |
| 9800 GTX | 128 | 512 | 648 | 140 | 1.1 |
| 9800 GTX+ | 128 | 2 * 512 | 705 | 141 | 1.1 |
| GTX 260 | 192 | 896 | 715,392 | 202 | 1.3 |
| GTX 275 | 240 | 896 | 1010.880 | 219 | x |
| GTX 285 | 240 | 1024 | 1062.720 | 204 | 1.3 |
| GTX 280 | 240 | 1024 | 933.120 | 236 | 1.3 |
| GTX 295 | 2 * 240 (2 GPUs) | 2 * 896 | 1788.480 | 289 | 1.3 |

| Modèle (Tesla) | Nb total de cœurs | Taille de mémoire (Mo) | GFLOPs | Consommation maximale (W) | Version de GPU (Compute capability) |
|---|---|---|---|---|---|
| C870 | 128 | 1 * 1536 | 519 | 170 | 1.0 |
| D870 | 2 * 128 (2 GPUs) | 2 * 1536 | 1037 | 520 | 1.0 |
| S870 | 4 * 128 (4 GPUs) | 4 * 1536 | 2074 | 800 | 1.0 |
| C1060 | 240 | 1 * 4096 | 933,12 (Single P.) 77,76 (Double P.) | 200 | 1.3 |
| C2050 | 448 | 2 * 1536 (ou 4 * 1536 - mémoire ECC) | 1288 (Single P.) 515,2 (Double P.) | 247 | 2.0 |
| C2070 | 448 | 2 * 1536 (ou 4 * 1536 - mémoire ECC) | 1288 (Single P.) 515,2 (Double P.) | 225 | 2.0 |

| Modèle (Quadro) | Nb de coeurs | Taille de mémoire (Mo) | GFLOPs | Puisssance dissipée totale | Version de GPU (Compute capability) |
|---|---|---|---|---|---|
| FX 580 | 32 | 512 | 108 | 40 | x |
| FX 1700 | 32 | 512 | 88,32 | 42 | 1.1 |
| FX 1800 | 64 | 768 | 264 | 59 | x |
| FX 4600 | 96 | 768 | 345 | 134 | 1.0 |
| FX 3700 | 112 | 512 | 420 | 78 | 1.1 |
| FX 3800 | 192 | 1024 | 693.504 | 108 | x |
| FX 4800 | 192 | 1536 | 693.504 | 150 | 1.3 |
| FX 5800 | 240 | 4096 | 933.12 | 189 | 1.3 |

Tableau 21 : Spécifications techniques par version de processeur graphique

| | Compute Capability | | | | |
|---|---|---|---|---|---|
| **Technical Specifications** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x** |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 |
| Maximum x-, y-, or z-dimension of a grid of thread blocks | 65535 | | | | |
| Maximum dimensionality of thread block | 3 | | | | |

Tesla G80 Tesla GT200 Fermi

| | Compute Capability | | | | |
|---|---|---|---|---|---|
| **Technical Specifications** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x** |
| Maximum x- or y-dimension of a block | 512 | | | | 1024 |
| Maximum z-dimension of a block | 64 | | | | |
| Maximum number of threads per block | 512 | | | | 1024 |
| Warp size | 32 | | | | |
| Maximum number of resident blocks per multiprocessor | 8 | | | | |
| Maximum number of resident warps per multiprocessor | 24 | | 32 | | 48 |
| Maximum number of resident threads per multiprocessor | 768 | | 1024 | | 1536 |
| Number of 32-bit registers per multiprocessor | 8 K | | 16 K | | 32 K |
| Maximum amount of shared memory per multiprocessor | 16 KB | | | | 48 KB |
| Number of shared memory banks | 16 | | | | 32 |
| Amount of local memory per thread | 16 KB | | | | 512 KB |
| Constant memory size | 64 KB | | | | |
| Cache working set per multiprocessor for constant memory | 8 KB | | | | |
| Maximum number of instructions per kernel | 2 million | | | | 512 million |

Tesla G80

Tesla GT200

Fermi

# Annexe 2
# Installation de CUDA™ et du pilote CUDA

## 1. Présentation

Pour pouvoir utiliser CUDA™ sur un système, ce dernier doit comporter :

- un processeur graphique compatible CUDA,
- Microsoft Windows XP, Vista ou 7 ou Windows Server 2003 ou 2008,
- un pilote de périphérique graphique,
- le logiciel CUDA (disponible gratuitement sur le site http://www.nvidia.com/cuda),
- Microsoft Visual Studio 2005 ou 2008, ou 2010 ou les versions correspondantes de Microsoft Visual C++ Express.

L'installation de CUDA™ sur un système d'exploitation Microsoft Windows se déroule en 4 étapes principales :

- vérifier que le système comporte un processeur graphique compatible CUDA,
- télécharger le logiciel CUDA,
- installer le pilote pour Windows XP (ou Windows Vista ou Windows 7),
- installer le logiciel CUDA.

## 2. Détail des 4 étapes de l'installation du logiciel CUDA :

### 2.1 Vérification de la compatibilité du périphérique de traitement graphique

Aujourd'hui, de nombreux produits NVIDIA contiennent des processeurs graphiques compatible CUDA. Parmi ces produits, on trouve principalement :

- les processeurs graphiques des séries NVIDIA GeForce® 8, 9 et 200, 400 et 500,
- les solutions de calcul NVIDIA Tesla,
- beaucoup de produits NVIDIA Quadro®.

Le processeur graphique présent sur le PC en début de projet (GeForce 8400 GS) faisait partie de la série 8, il était donc compatible CUDA.

### 2.2 Téléchargement du logiciel CUDA

Le pilote CUDA est intégré dans le pilote graphique NVIDIA qui peut être téléchargé sur le site : http://www.nvidia.com/object/cuda_get.html

### 2.3 Installation du pilote pour Windows XP

Pour utiliser CUDA Toolkit, il est nécessaire d'avoir au moins la version de pilote graphique NVIDIA spécifiée dans le document *CUDA Toolkit Release Notes* le plus récent.

Pour connaître la version du pilote NVIDIA installé, il suffit de regarder dans le panneau de configuration NVIDIA. Ouvrir le panneau de configuration NVIDIA en cliquant droit sur le bureau et en sélectionnant *Panneau de configuration NVIDIA* (voir figure suivante) puis *Informations système* en bas à gauche du panneau de configuration principal.

Dans le PC utilisé pour le projet, la version de départ du pilote graphique était 182.50, cette version a donc du être changée plusieurs fois. Ce pilote a été mis à jour progressivement par les versions 191.07, 260.99, 266.58.

A noter que les nouvelles versions du Toolkit CUDA nécessitent typiquement les nouvelles versions du pilote NVIDIA également, donc il est préférable de toujours vérifier que la bonne version de pilote est présente pour la version du Toolkit CUDA utilisée.

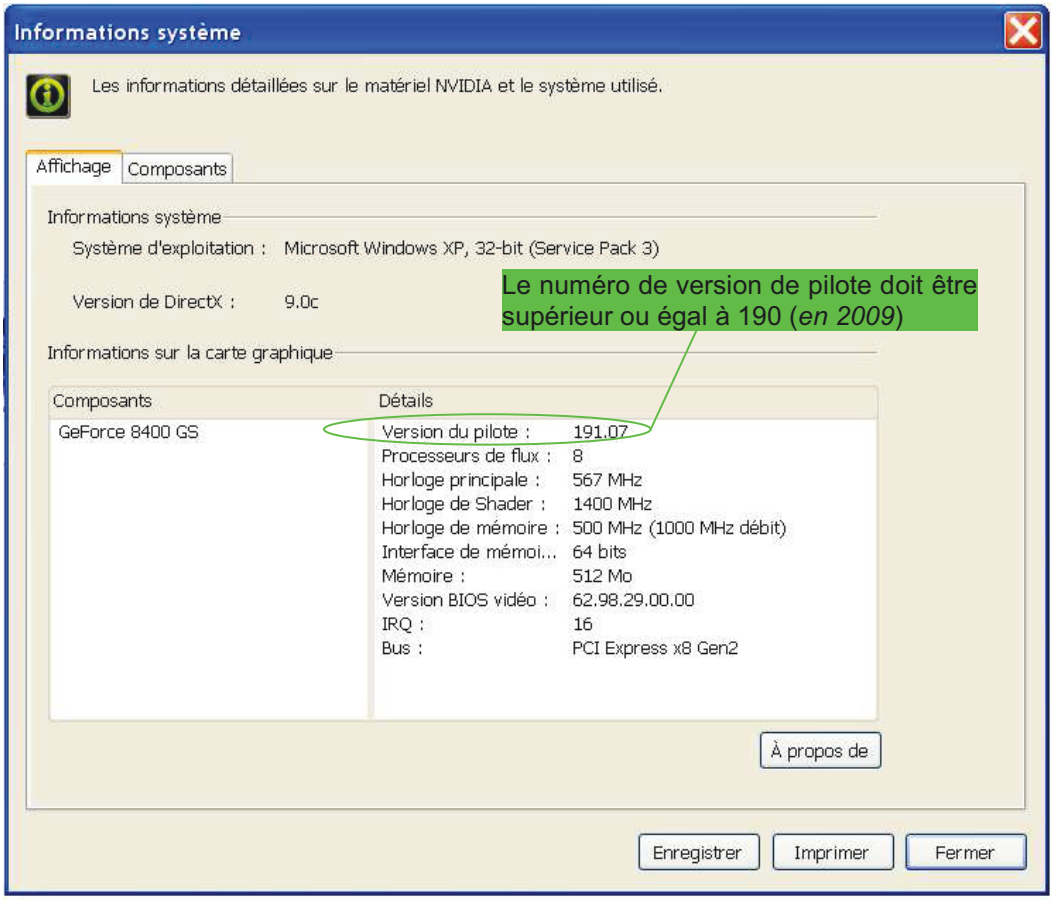
## 2.4 Installation du logiciel CUDA

### 2.4.1 Présentation

Pour faire fonctionner des programmes CUDA, il faut avoir le matériel CUDA suivant :

- le *CUDA Toolkit*,
- le kit de développement (*SDK*) CUDA.

Le *CUDA Toolkit* contient les outils nécessaires pour compiler et concevoir une application CUDA en conjonction avec Microsoft Visual Studio. Il inclut des outils, des bibliothèques, des fichiers en-tête et d'autres ressources.




Le kit de développement (*SDK*) CUDA inclut :

- des projets échantillons qui ont toute la configuration projet nécessaire,
- des fichiers de génération pour réaliser des générations un-clic utilisant Microsoft Visual Studio.

Les logiciels sont disponibles pour Windows 32-bits et Windows 64-bits.

Pour une installation réussie, il est conseillé de suivre la procédure suivante :

2.4.2 Télécharger le logiciel CUDA™ NVIDIA à partir du site http://www.nvidia.com/object/cuda_get.html et sauvegarder l'installateur sur le bureau.

2.4.3 Désinstaller les versions précédentes du *CUDA Toolkit* et du kit de développement CUDA™ NVIDIA s'ils étaient déjà installés.

Pour désinstaller le *CUDA Toolkit*, passer par le menu *Démarrer* de la manière suivante : *Démarrer → Tous les programmes → NVIDIA Corporation → Cuda Toolkit → Uninstall CUDA*

Pour désinstaller le kit de développement CUDA, un chemin similaire est utilisé : *Démarrer → Tous les programmes → NVIDIA Corporation → NVIDIA GPU Computing SDK → uninstall NVIDIA GPU Computing SDK*

2.4.4 Installer le *CUDA Toolkit* en exécutant son paquetage d'installation et en suivant les instructions apparaissant à l'écran. Le *CUDA Toolkit* est installé par défaut sous *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#* avec *#.#* représentant la version 3.2 ou plus. Ce répertoire contient ce qui suit :

- *CUDA_BIN_PATH* qui est une variable d'environnement représentant le chemin (se terminant par *CUDA\bin*) de l'emplacement contenant les bibliothèques du moteur d'exécution (*runtime*) et les exécutables du compilateur.
- *CUDA_INC_PATH* qui est une variable d'environnement représentant le chemin (se terminant par *CUDA\include* ou par *CUDA\inc*) de l'emplacement contenant les fichiers *include* (les fichiers *.h*) nécessaires pour compiler des programmes CUDA.
- *CUDA_LIB_PATH* qui est une variable d'environnement représentant le chemin (se terminant par *CUDA\lib*) de l'emplacement contenant les bibliothèques (les *.lib*) nécessaires pour l'édition de lien des codes CUDA.
- **\CUDA.doc* qui est le chemin de l'emplacement de la plupart des documentations CUDA en particulier le guide de programmation (*Programming Guide*).

Les variables d'environnement ci-dessus seront utilisées en particulier au moment de la configuration du projet et du fichier contenant le code.

2.4.5 Installer une nouvelle version du kit de développement CUDA™ NVIDIA en exécutant *cudasdk_#.#_win_32.exe* après avoir remplacé *#.#* par le numéro de version à installer. Le kit de développement CUDA est installé sous *C:\Documents and Settings\All Users\Application Data\NVIDIA Corporation\NVIDIA GPU Computing SDK* et contient le code source pour de nombreux problèmes et modèles exécutables sous Microsoft Visual Studio.

# Annexe 3
# Installation et configuration de Visual Studio 2005

## 1. Installation de Visual Studio 2005

Microsoft Visual Studio est disponible dans les éditions suivantes :

- Visual Studio Team System,
- Visual Studio Professional,
- Visual Studio Standard,
- Visual Studio Express.

Microsoft Visual Studio Express est un modèle d'environnement de développement intégré gratuit développé par Microsoft. Il s'agit d'une version allégée de Microsoft Visual Studio. Le choix du logiciel de développement à utiliser pour CUDA dans le cadre de ce mémoire s'est porté sur la version Express de Visual Studio 2005 car elle fournit un environnement de développement facile à utiliser.

La version de Visual Studio Express (2005) est sortie le 7 novembre 2005 et était supposée n'être gratuite que pour un an (mis à part SQL Server 2005 Express Edition). Toutefois, Microsoft a annoncé le 19 avril 2006 que ces éditions resteront gratuites à vie.

Microsoft Visual Studio Express est composé des différents produits suivants, chacun supportant un seul langage :

- Visual Basic
- C#
- C++
- J#
- Web Developer
- SQL Server

Tableau 22 : Configuration requise pour installer Visual Studio Express

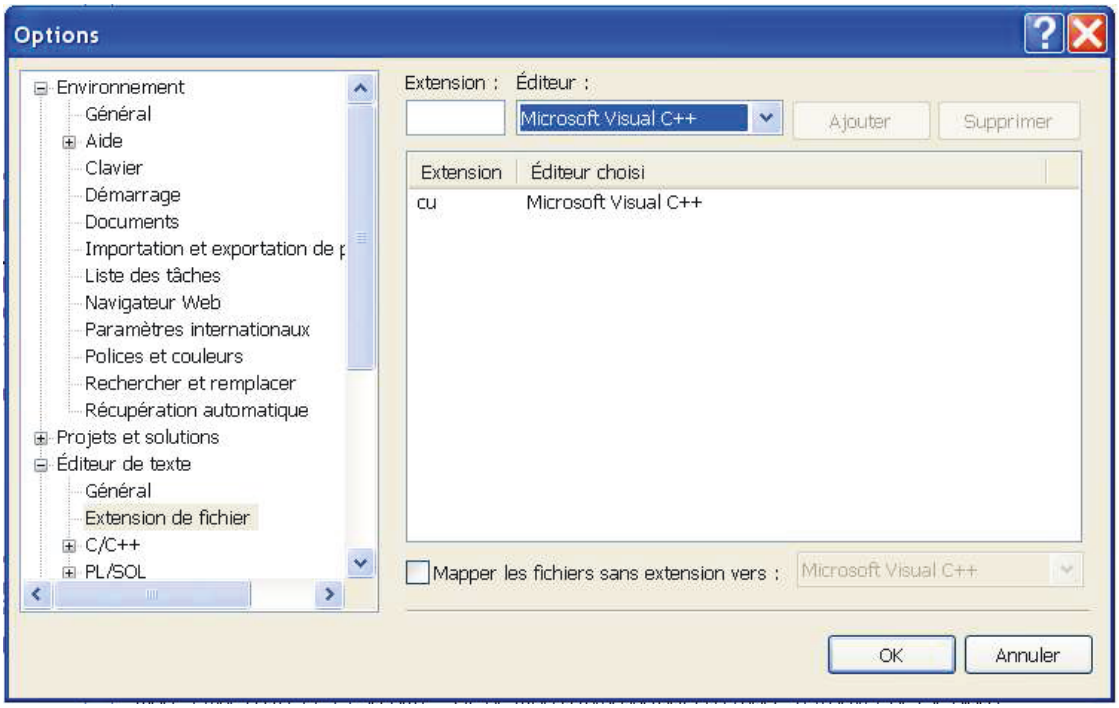
| | **Configuration requise pour installer Visual C++** | **Configuration du PC utilisé** |
|---|---|---|
| Processeur | Type Pentium III, 600 MHz<br>Recommandé : 1 GHz | Q9650, 3 GHz |
| Mémoire vive | 192 Mo<br>Recommandé : 256 | 3.25 Go |
| Système d'exploitation | Windows 2000, Windows XP ou Windows Server 2003 | Windows XP |
| Espace disponible sur disque dur | Capacité de base :<br>• 40 Mo d'espace disponible requis sur le lecteur système<br>• 160 Mo d'espace disponible requis sur le lecteur d'installation | 46.7 Go disponible |

On constate qu'il n'y a pas de problème de compatibilité entre Visual Studio Express et le PC utilisé pour ce mémoire, les propriétés du PC étant bien supérieures à celles requises pour installer le logiciel.

## 2. Configuration de Visual Studio pour le développement et l'exécution d'un programme MEX-CUDA

Avant toute configuration de code et de projet dans Visual Studio, un fichier de code CUDA (*.cu*) est affiché par défaut sans la coloration syntaxique disponible habituellement dans un code C, ce qui rend l'édition du programme assez délicate. Pour retrouver cette coloration syntaxique, NVIDIA fournit le fichier *usertype.dat* qui peut se trouver sous: *C:\Documents and Settings\All Users\Application Data\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\doc\syntax_highlighting\visual_studio_8*

- Copier ce fichier dans *C:\Program Files\Microsoft Visual Studio\Common\IDE\visual_studio_8*
- Dans Visual Studio, aller dans le menu *Outils* → *Options* → *Editeur de texte* → *Extension de fichier*. Dans le champ *Extension*, taper *cu*, vérifier que Microsoft Visual C++ est bien sélectionné dans la liste déroulante de *Editeur*. Cliquer ensuite sur *Ajouter* puis sur *OK*.

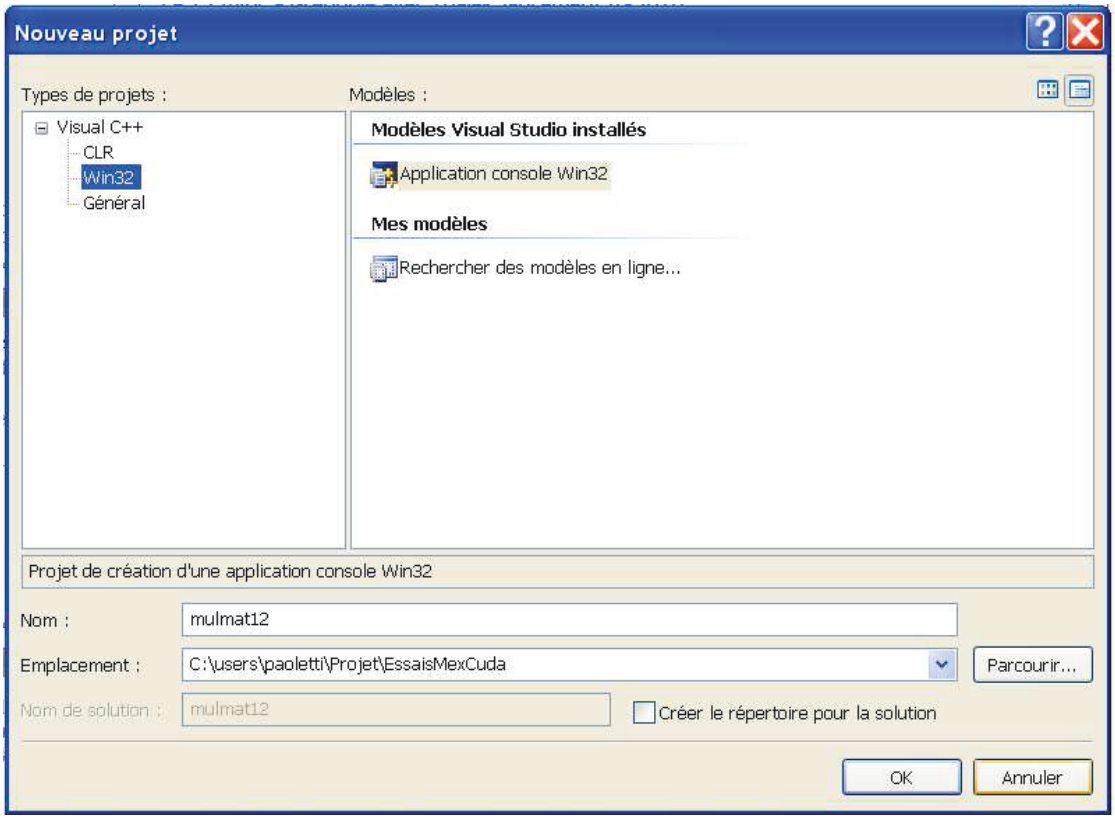



Contrairement aux propriétés d'un projet qui doivent être redéfinies à chaque nouveau projet, la configuration de la coloration syntaxique est permanente et ne nécessite aucune nouvelle configuration lors d'un nouveau projet car c'est Visual Studio lui-même qui est configuré (et non pas le fichier de code CUDA ou le projet CUDA).

# Annexe 4
# Création et configuration d'un projet CUDA™, CUBLAS ou CUFFT sous Visual Studio 2005

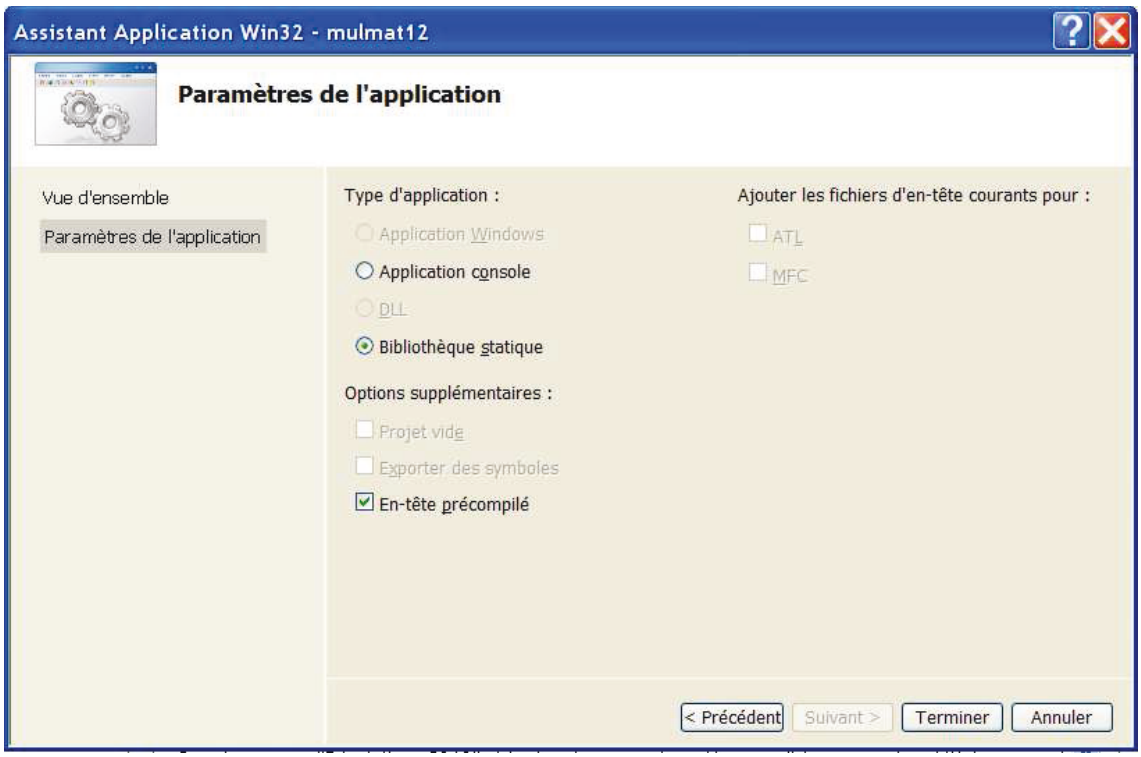
## 1. Création d'un nouveau projet MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

- Ouvrir Visual Studio 2005.
- Pour créer un nouveau projet, aller dans les menus et sous-menus suivants : *Fichier → Nouveau → Projet*.
- Si ce n'est pas fait, cliquer sur *Win 32* et *Application console Win 32* pour les sélectionner.
- Donner un nom pour le projet dans le champ *Nom*.
- Choisir l'emplacement – dans lequel sera stocké le dossier contenant tous les fichiers du projet – dans le champ *Emplacement*.
- Vérifier que la case *Créer le répertoire pour la solution* n'est pas cochée.
- Cliquer sur *OK*.




- Dans la fenêtre qui s'ouvre, cliquer sur *suivant*.


- Dans la fenêtre qui s'ouvre, choisir *Bibliothèque statique* dans la rubrique *Type d'application* (qui devra être changé en *dll* au moment de la configuration de projet).
- Vérifier que la case *En-tête précompilé* est cochée dans la rubrique *Options supplémentaires*.
- Cliquer sur *Terminer*.




Cette procédure de création de projet est identique pour les projet MEX-CUBLAS et MEX-CUFFT.

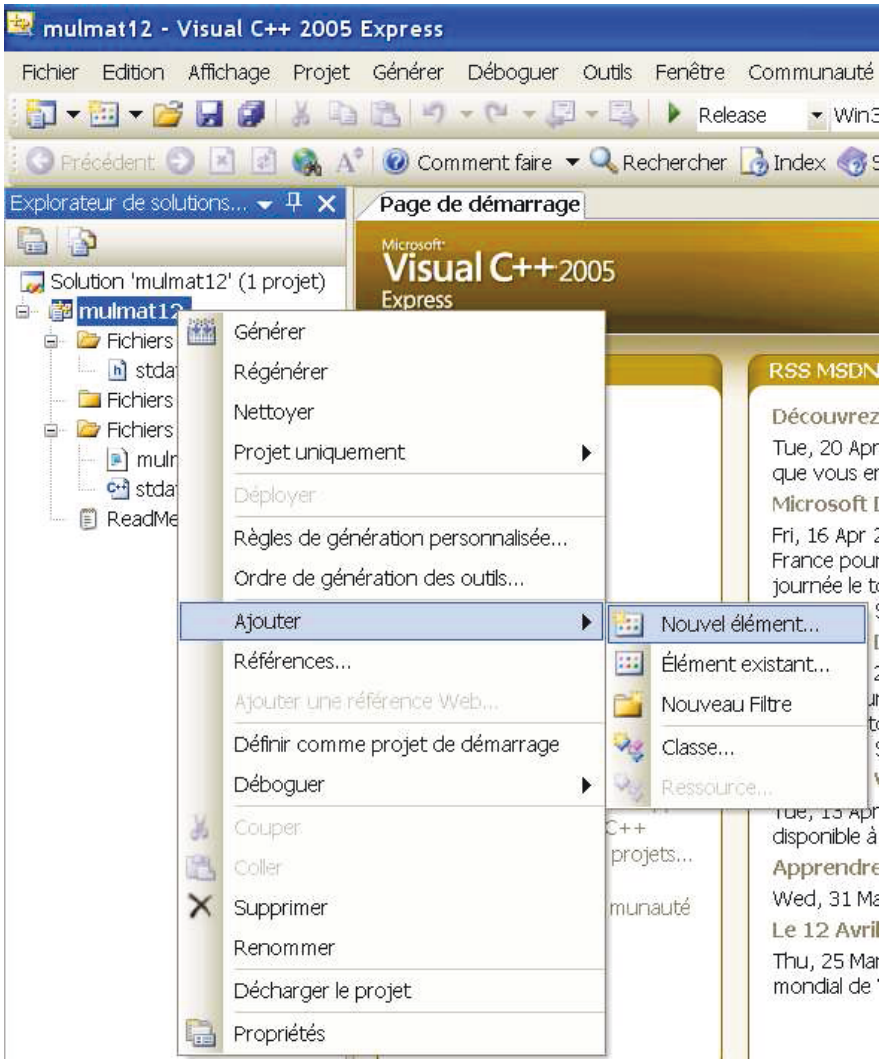
## 2. Création d'un fichier de définition de module MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

Pour pouvoir exécuter le code, Matlab a besoin de connaître le « *point d'entrée* ». Dans le cas des fichiers MEX, Matlab essaiera toujours d'exécuter une fonction appelée « `mexFunction` ». Par conséquent, il est nécessaire d'informer le compilateur que cette fonction doit être accessible de l'extérieur. Il faut donc créer un fichier de définition de module dont le rôle est de donner au compilateur le nom des fonctions qui seront accessibles de l'extérieur.

Les fichiers de définition de module (*.def*) fournissent à l'éditeur de liens des informations sur les exportations. Un fichier *.def* est surtout utile lors de la génération d'une *dll*.

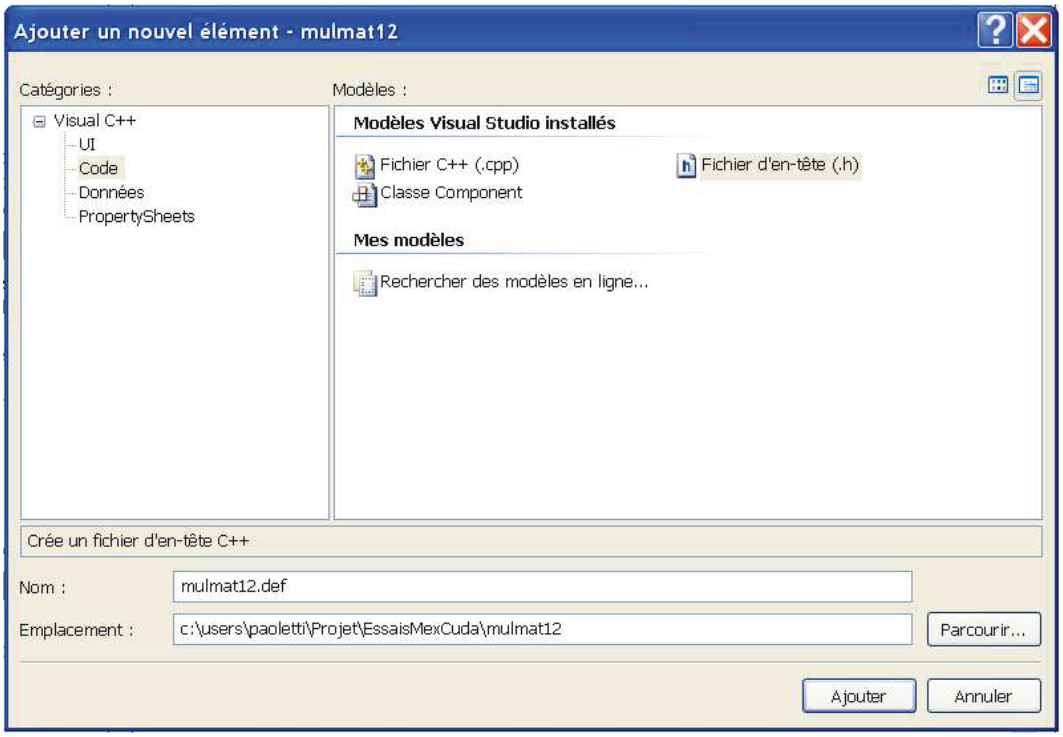
Créer d'abord un nouveau fichier vide avec l'extension *.def*.

- Le plus simple est de cliquer droit sur le nom du projet (*mulmat12*) puis d'aller dans le menu *Ajouter → Nouvel élément…*




- Puis cliquer sur *Code* dans le menu *Catégories* et sur *Fichier d'en-tête (.h)* dans le menu *Modèles*.
- Donner un nom au fichier avec une extension *.def* dans le champ *Nom*.

- Vérifier que l'emplacement est bien la racine du projet, c'est-à-dire le répertoire contenant tous les fichiers du projet.
- Cliquer sur *Ajouter*.




- Dans le fichier texte qui s'ouvre, écrire les deux lignes suivantes :
  - *LIBRARY mulmat12*
  - *EXPORTS mexFunction*
  - Remplacer le nom de fichier MEX *mulmat12* par un nom en rapport avec le programme à développer.
- Sauvegarder le fichier.
- Fermer le fichier de définition de module.

Le mot clé *LIBRARY* suivi du nom *mulmat12* indique à l'éditeur de lien de créer une *dll* dont le nom sera *mulmat12*. Parallèlement, l'éditeur de lien crée une bibliothèque d'importation. Le nom *mulmat12* représente le nom qui sera utilisé pour appeler la fonction à partir de Matlab.

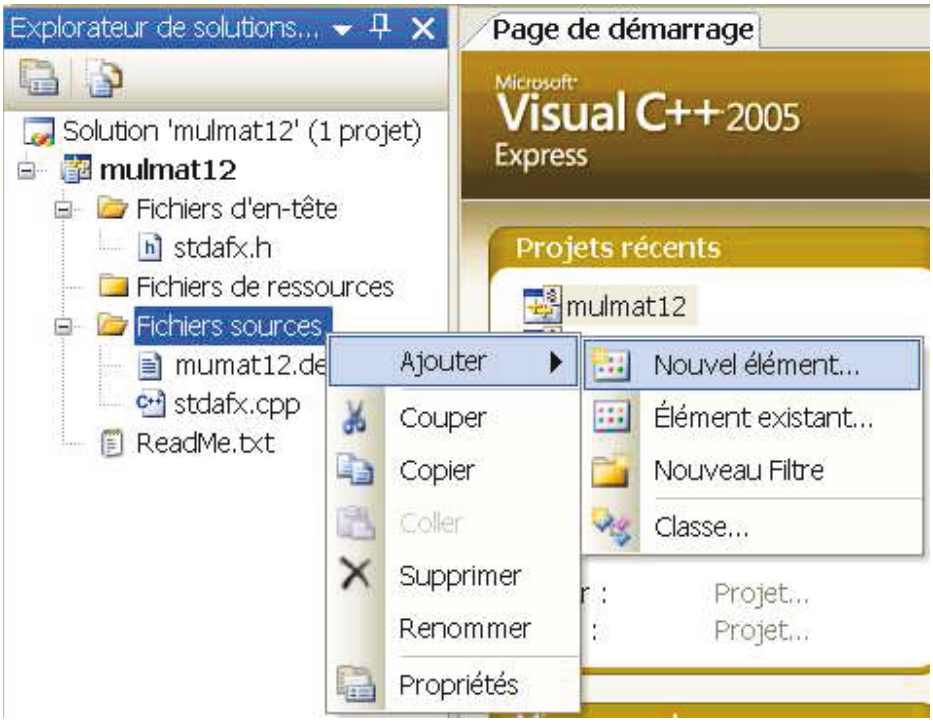
Le mot clé *EXPORTS* introduit une section qui contient une ou plusieurs *definitions* représentant des données ou des fonctions exportées. Dans notre cas, il n'y a qu'une définition représentant une fonction : *mexFunction*.

Cette procédure de création de fichier de définition de module est identique pour les projet MEX-CUBLAS et MEX-CUFFT.

## 3. Création d'un fichier code MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

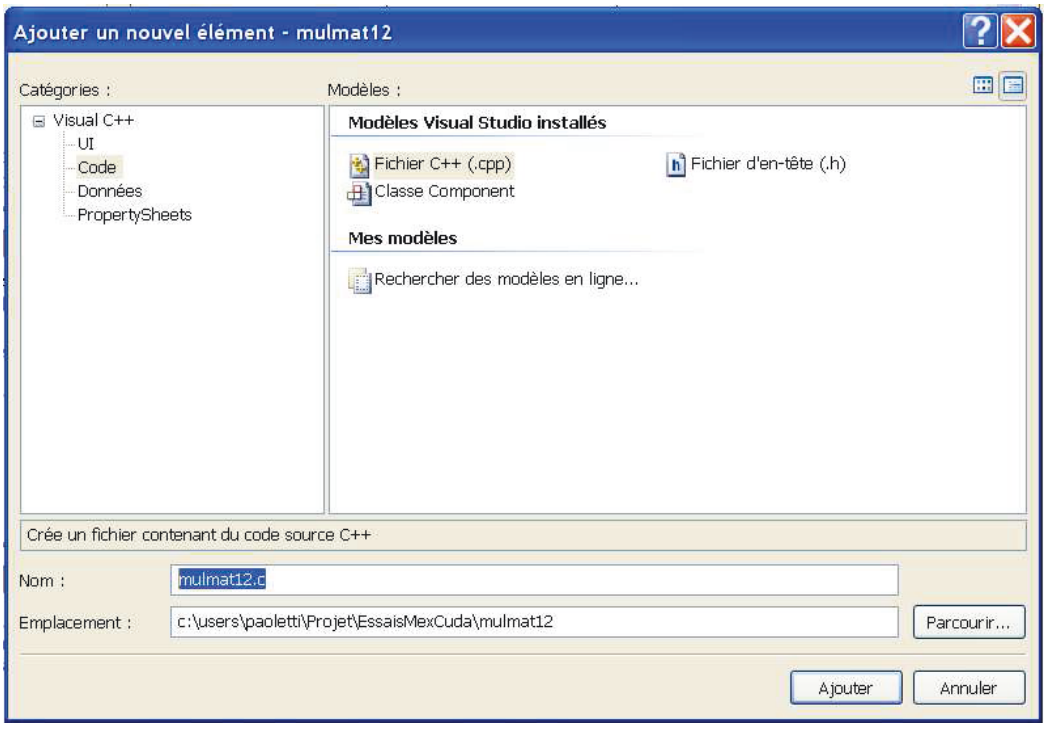
Pour créer le fichier qui contiendra le code, cliquer avec le bouton droit de la souris sur *Fichiers sources* dans l'*Explorateur de solutions* de VS2005 puis sur les menus et sous-menus suivants : *Ajouter → Nouvel élément*

Cette procédure de création de fichier code MEX-CUDA est identique pour les projet MEX-CUBLAS et MEX-CUFFT.




Dans la fenêtre qui s'ouvre :

- Cliquer sur la catégorie *Code* pour la sélectionner (si elle ne l'est pas déjà).
- Cliquer sur le modèle *Fichier C++ (.cpp)* pour le sélectionner (s'il ne l'est pas déjà).



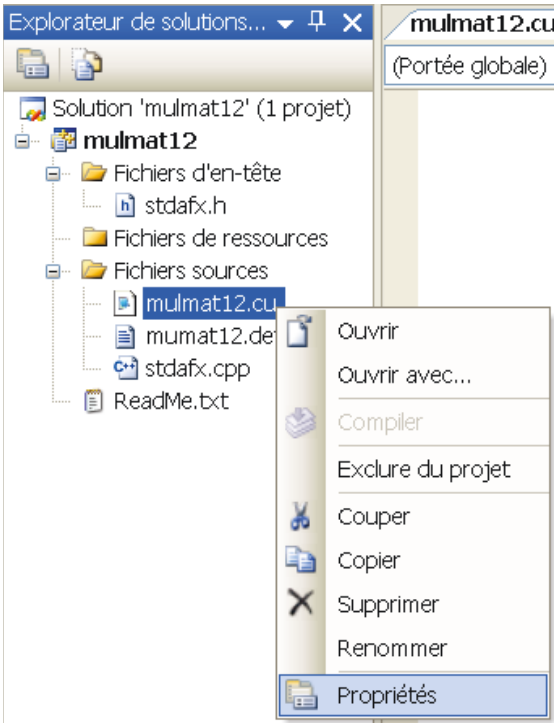

- Entrer le nom du fichier qui contiendra le programme suivi de l'extension *.cu* dans le champ *Nom*.
- Donner l'emplacement où se trouvera le fichier dans le champ *Emplacement*.
- Cliquer sur *Ajouter*.

# 4. Configuration d'un fichier MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

## 4.1 Procédure de configuration

Une fois le fichier du code créé, il doit être configuré. Dans un premier temps, il faut donc configurer le fichier source contenant le code :

- Cliquer droit sur le nom du fichier (*mulmat12.cu*) dans l'*explorateur de solutions* puis sur *Propriétés*.




Dans la fenêtre de propriétés, un mode de configuration doit d'abord être sélectionné avant de remplir les différents champs.

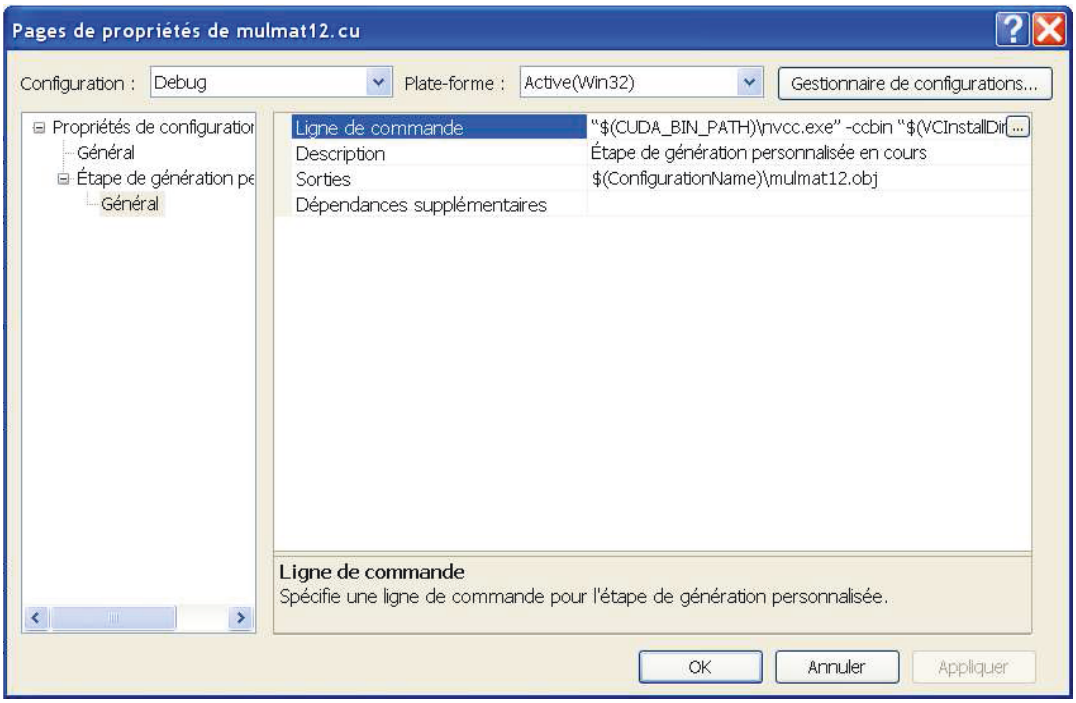
La configuration en mode *Debug* permet de corriger facilement d'éventuelles erreurs de programme mais offre une exécution moins rapide que la configuration en mode *Release* (qui elle ne permet pas de faire d'exécution pas à pas avec points de contrôle).

Cette procédure de configuration du fichier code CUDA est identique pour les projet MEX-CUBLAS et MEX-CUFFT.

**Attention** : Dans la fenêtre de modifications des propriétés du programme, toujours vérifier quel mode de configuration (*Debug* ou *Release*) est sélectionné car à chaque ouverture de la fenêtre de propriétés, c'est toujours le mode actif qui est sélectionné par défaut, ce qui peut être source de confusion. Pendant le déroulement des différentes étapes de configuration, il est également très conseillé de cliquer sur l'icône *Appliquer* systématiquement après chaque configuration dans un mode donné.

- Aller dans les menus et sous-menus suivants : *Propriétés de configuration → Etape de génération personnalisée → Général* puis dans le champ *Ligne de commande*, ajouter les lignes suivantes :
    - En mode Debug : *"$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)\bin" -arch=sm_20 --ptxas-options=-v -c -Xcompiler /EHsc,/W3,/nologo,/Wp64,/Od,/ZI,/MDd -I"$(CUDA_INC_PATH)" -I./ -I"C:\MATLAB\R2007b\extern\include" -maxrregcount=24 -o $(ConfigurationName)\mulmat12.obj mulmat12.cu*
    - En mode Release : *"$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)\bin" -arch=sm_20 -c -Xcompiler /EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MD -I"$(CUDA_INC_PATH)" -I./ -I"C:\MATLAB\R2007b\extern\include" -maxrregcount=24 -o $(ConfigurationName)\mulmat12.obj mulmat12.cu*
- Pour ces deux configurations, changer le nom *mulmat12* par le nom du fichier contenant le code.
- Pour continuer la configuration du fichier, sans changer de menus et sous-menus, sélectionner le mode *Toutes les configurations* puis entrer la ligne *$(ConfigurationName)\mulmat12.obj* dans le champ *Sorties*. Cet ajout permet de spécifier la sortie dans laquelle sera généré le fichier objet.

Changer le nom *mulmat12* par le nom du fichier contenant le code.




## 4.2 Description détaillée de la configuration du projet

La syntaxe spécifiée par *$(nom)* permet de référer la chaîne de texte désignée à la valeur courante de variables d'environnements.

Le terme *CUDA_BIN_PATH* est donc une variable d'environnement correspondant à un chemin donné à Visual C++ lui indiquant l'emplacement du compilateur CUDA (NVCC) pour pouvoir l'exécuter. Ce terme correspond à un chemin se terminant par : *\CUDA\bin* c'est à dire dans le cadre de ce projet : *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\bin*. Le chemin exact contenu dans *CUDA_BIN_PATH* se trouve en suivant la procédure décrite plus bas (Annexe 4 §4.3).

Les options du compilateur sont spécifiées par une barre oblique (/) ou un tiret (-)

La commande *-ccbin* permet de spécifier le répertoire dans lequel l'exécutable du compilateur hôte réside. Par défaut cet exécutable est dans le chemin de recherche courant de l'exécutable.

La variable d'environnement *VCInstallDir* désigne l'emplacement de Visual C++ dans l'ordinateur, elle correspond donc au chemin *C:\Program Files\Microsoft Visual Studio 8\VC*. Le chiffre 8 de ce chemin correspond au numéro de version interne de Visual Studio 2005. Toutes les bibliothèques utilisées par Visual Studio se trouvent dans ce dossier. Le chemin exact contenu dans *VCInstallDir* se trouve en suivant la procédure décrite plus bas (Annexes 4 §4.3).

Le terme *-arch* sert à spécifier le type d'architecture de processeur graphique pour lequel la compilation va se faire. Cette architecture peut être soit « réelle » soit « virtuelle » (*ptx*). Le code *ptx* représente un format intermédiaire qui peut encore être compilé et optimisé pour une classe spécifique de processeurs graphiques réels (selon la version *ptx*). L'architecture spécifiée par cette option est l'architecture qui est adoptée par la chaîne de compilation jusqu'à l'étape *ptx*, alors que l'architecture (ou les architectures) spécifiée(s) par l'option *–code* est (sont) adoptée(s) par la dernière étape de compilation, donc potentiellement l'étape du moteur d'exécution.

Les architectures de compilation prise en charge actuellement (2010) sont :

- les architectures virtuelles *compute_10*, *compute_11*, *compute_12*, *compute_13* ;
- les architectures GPU *sm_10*, *sm_11*, *sm_12* et *sm_13*.

Les numéros (10, 11, 12 et 13) correspondent à la version (*Compute Capability*) de périphériques de calcul compatible CUDA. Etant donné que les programmes de ce projet sont uniquement destinés à être exécutés sur carte GTX470 (Version : 2.0), il faut spécifier au compilateur de compiler pour une architecture GPU réelle de version 2.0. D'où l'ajout de *-arch=sm_20* dans la ligne de commande. Cette option est utile dans le cas – tel que dans cette étude de CUDA™ – où on veut faire des calculs employant des nombres flottants double précision. Si cette option n'est pas utilisée pour un programme prévu avec des *double*, on obtient le warning *Double is not supported. Demoting to float*. C'est-à-dire que par défaut, la double précision n'est pas prise en charge et est convertie en simple précision avant le calcul.

Le terme *--ptxas-options* sert à spécifier des options directement à l'assembleur d'optimisation *ptx*. Le terme *–v* est une option qui sert à lister les commandes de compilation générées par le pilote de compilateur, mais ne supprime par leur exécution. Cette option est particulièrement utile pour connaître le nombre de registres employés lors de la phase de développement et de débogage (donc surtout en mode *Debug*) . Cette option n'a plus d'utilité en mode *Release*.

La commande *-c* correspond à une phase de compilation demandant au compilateur de transformer le fichier *c* en fichier objet. Le suffixe du nom du fichier source étant remplacé par *obj*.

Le terme *–Xcompiler* permet de spécifier des options directement au compilateur.

Le terme */EHsc* est constitué de deux parties :

- Le terme */EH*
- Les termes *s* et *c* appelés arguments

Le terme */EH* spécifie le modèle de gestion des exceptions à utiliser par le compilateur et détruit les objets C++ qui sont hors de portée suite à l'exception.

Le terme *s* représente le modèle de gestion des exceptions qui intercepte uniquement les exceptions C++ et indique au compilateur de présumer que les fonctions C de type *extern* lèvent une exception.

Le terme *c*, s'il est utilisé avec *s* (*/EHsc*) intercepte uniquement les exceptions C++ et indique au compilateur de présumer que les fonctions C de type *extern* ne lèvent jamais une exception C++.

Le terme */W3* fixe le niveau d'avertissement généré par le compilateur à 3. La fourchette d'avertissement valide s'étend de 0 à 4 :

- le niveau 0 désactive tous les avertissements,
- le niveau 1 affiche les avertissements graves. Le niveau 1 est le niveau d'avertissement par défaut pour la ligne de commande,
- le niveau 2 affiche tous les avertissements de niveau 1 et les avertissements moins graves que le niveau 1,
- le niveau 3 affiche tous les avertissements de niveau 2 et tous les autres avertissements recommandés pour des objectifs de production,
- le niveau 4 affiche tous les avertissements de niveau 3 ainsi que les avertissements informationnels qui peuvent être ignorés sans risque dans la plupart des cas.

Le terme */nologo* supprime :

- l'affichage de la bannière de signe au démarrage du compilateur,
- l'affichage de messages informationnels pendant la compilation.

Le terme */Wp64* détecte les problèmes de portabilité 64-bits. Ce terme est :

- désactivé par défaut dans le compilateur 32-bits de Visual C++,
- activé par défaut dans le compilateur 64-bits de Visual C++.

Le terme */Od* désactive l'optimisation. Il désactive toutes les optimisations du programme et accélère la compilation. Cette option est une option par défaut.

Le terme */O2* est une option qui permet de réduire la taille du code ou d'augmenter la vitesse. Cette option augmente la vitesse en créant le code le plus rapide dans la majorité des cas (paramètre par défaut pour les versions release).

Le terme */ZI* produit une base de données du programme, dans un format qui prend en charge la fonctionnalité *Modifier & Continuer*. Cette option est nécessaire pour utiliser le débogage *Modifier & Continuer*.

*Modifier & Continuer* est une fonctionnalité qui fait gagner du temps en permettant de modifier le code source pendant que le programme est en mode arrêt. Lorsqu'on reprend l'exécution du programme en choisissant une commande d'exécution telle que *Continue* ou *Step*, la fonctionnalité *Modifier & Continuer* applique automatiquement les modifications du code, avec certaines limitations toutefois. Cela permet de modifier le code pendant la session de débogage, au lieu de devoir arrêter, recompiler le programme entier et redémarrer la session de débogage.

Le terme */Zi* génère des informations de débogage complètes. Ce terme produit une base de données qui contient des informations de *type* et des informations symboliques de débogage à utiliser avec le débogueur. Les informations symboliques de débogage incluent :

- les noms,
- les types de variable,
- les fonctions,
- les numéros de ligne.

Ce terme n'affecte pas les optimisations. Cependant, */Zi* implique d'avoir choisi l'option */DEBUG*. Les informations de type sont placées dans le fichier *.pdb* et non pas dans le fichier *.obj*.

L'option */Zi* du compilateur sert à stocker les informations de débogage du fichier *.obj* dans une base PDB. L'éditeur de liens recherche d'abord la base PDB de l'objet dans le chemin d'accès absolu écrit dans le fichier *.obj*, puis dans le répertoire contenant le fichier *.obj*.

L'option */MDd* définit *_DEBUG*, *_MT* ainsi que *_DLL*, et indique à l'application d'utiliser les versions spécifiques de débogage multitâches et *dll* de la bibliothèque du moteur d'exécution (*runtime*). Le compilateur place aussi le nom de la bibliothèque *MSVCRTD.lib* dans le fichier *.obj*.

L'option */MD* indique à l'application d'utiliser les versions spécifiques multitâches et *dll* de la bibliothèque du moteur d'exécution. Cette option définit *_MT* ainsi que *_DLL*, et indique au compilateur de placer le nom de la bibliothèque *MSVCRT.lib* dans le fichier *.obj*.

L'option *–I* spécifie les chemins de recherche *include*. Le terme *–I* ajoute le répertoire indiqué par le chemin à la liste des répertoires. Ce répertoire est ajouté en tête de liste.

Le terme *-I"$(CUDA_INC_PATH)"* permet de dire à Visual C++ d'inclure la variable d'environnement *CUDA_INC_PATH* représentant le chemin de l'emplacement contenant les fichiers *include* de CUDA™ nécessaires au fonctionnement du programme. *CUDA_INC_PATH* correspond au chemin se terminant par : *\CUDA\include* ou *\CUDA\inc*, donc dans le cadre de ce projet : *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\include*. Le chemin exact contenu dans *CUDA_INC_PATH* se trouve en suivant la procédure décrite plus bas (Annexe 4 §4.3).

Le terme *-I"C:\MATLAB\R2007b\extern\include"* permet de dire à Visual C++ d'inclure le chemin *C:\MATLAB\R2007b\extern\include* pour lui permettre d'accéder aux fichiers *include* de Matlab nécessaires au fonctionnement du programme. Le terme *–I* ajoute le répertoire indiqué par le chemin *C:\MATLAB\R2007b\extern\include* à la liste des répertoires. Ce répertoire est ajouté en tête de liste. Ici, le terme *-I* spécifie un fichier de recherche *include*.

Le terme *–maxrregcount* est une option permettant de spécifier le nombre maximal de registres que les fonctions GPU peuvent utiliser. Une valeur plus grande (dans les limites spécifiques à une fonction) augmentera la performance de tâches GPU individuelles qui exécutent cette fonction. Cependant, une valeur plus élevée de cette option réduira également la taille de bloc (de tâches) maximale, réduisant ainsi la quantité de parallélisme des tâches parce que les registres de tâche sont alloués à partir d'un ensemble limité de registres sur chaque processeur graphique. C'est pourquoi, une bonne valeur de *maxrregcount* est le résultat d'un compromis. Si cette option n'est pas spécifiée, alors aucun maximum n'est pris en compte (dans la limite de 128 registres).

Le terme *-o* spécifie le nom et l'emplacement du fichier généré par l'éditeur de liens (fichier de sortie). Ce terme est donc suivi de *$(ConfigurationName)\mulmat12.obj* qui comporte *ConfigurationName* (le chemin), c'est-à-dire :

- soit : *C:\users\paoletti\Projet\EssaisMexCuda\mulmat12\Debug*
- soit : *C:\users\paoletti\Projet\EssaisMexCuda\mulmat12\EmuDebug*
- soit : *C:\users\paoletti\Projet\EssaisMexCuda\mulmat12\EmuRelease*
- soit : *C:\users\paoletti\Projet\EssaisMexCuda\mulmat12\Release*

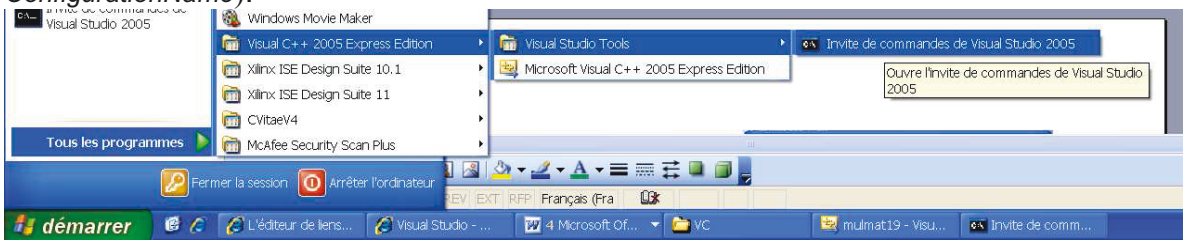
selon la configuration courante dans Visual C++.

Le terme *mulmat12.obj* représente le fichier généré à la compilation. Le terme suivant (*mulmat12.cu*) représente le fichier source utilisé pour générer le fichier objet. Le chemin exact contenu dans *ConfigurationName* se trouve en suivant la procédure décrite ci-dessous.

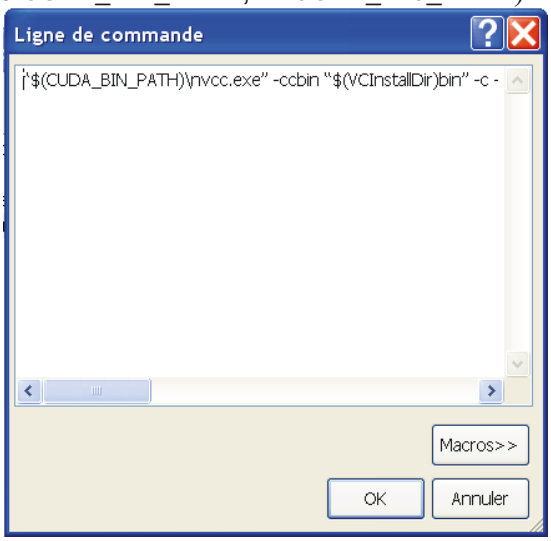
## 4.3 Procédure pour connaître le contenu d'une variable d'environnement

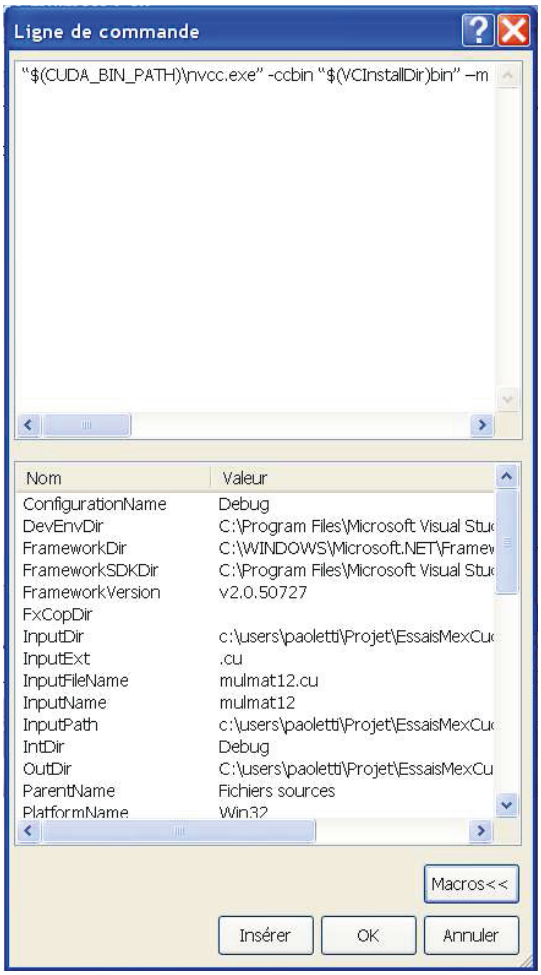
Selon le type de variable d'environnement, il existe deux façons de connaître leur contenu :

- soit en tapant *set* dans l'invite de commande de Visual Studio. L'invite de commande de Visual Studio se trouve en allant dans le menu : *Démarrer → Tous les programmes → Visual C++ 2005 Express Edition → Visual Studio Tools → Invite de commandes de Visual Studio 2005*. Parmi les nombreuses variables d'environnement, se trouvent en particulier *VCInstallDir*, *CUDA_BIN_PATH* et *CUDA_INC_PATH* (mais pas *ConfigurationName*).




- soit en cliquant sur l'icône « … » dans le champ *Ligne de commande* dans les propriétés du fichier source contenant le code, puis sur l'icône *Macros*. Parmi les nombreuses variables d'environnement, se trouvent en particulier *VCInstallDir* et *ConfigurationName* (mais pas *CUDA_BIN_PATH*, ni *CUDA_INC_PATH*).





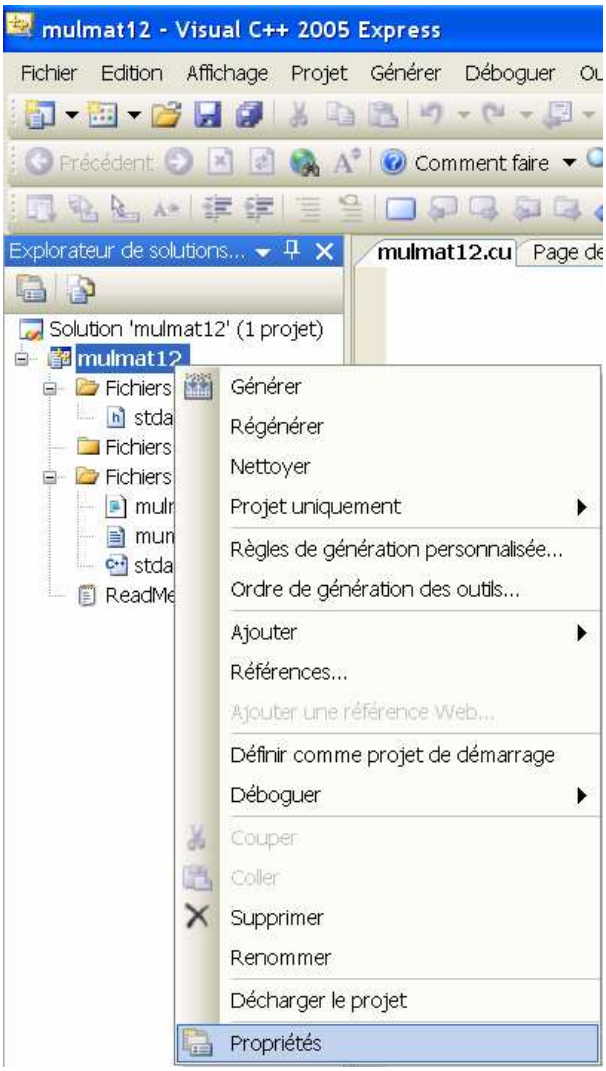

# 5. Configuration d'un projet MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

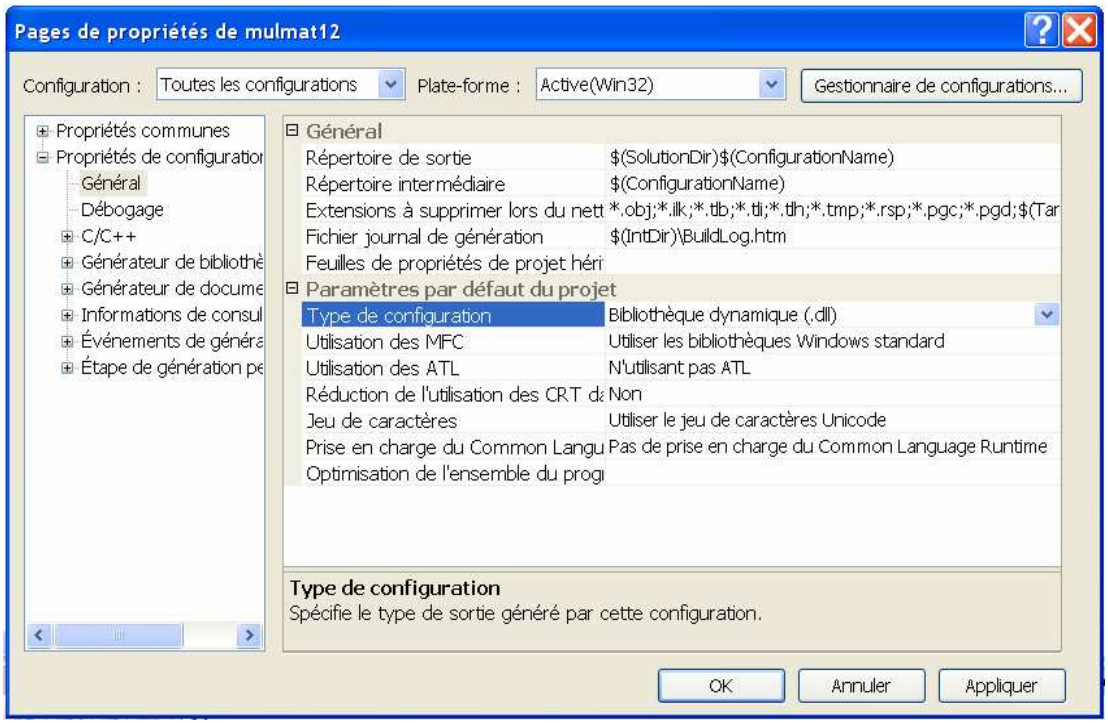
## 5.1 Procédure de configuration

Une fois le fichier code configuré, il faut aussi configurer le projet. Pour cela :

- cliquer droit sur le nom du projet, puis cliquer sur *Propriétés*,
- dans la fenêtre qui s'ouvre représentant les propriétés du projet, choisir *Toutes les configurations* dans le champ *Configuration*.

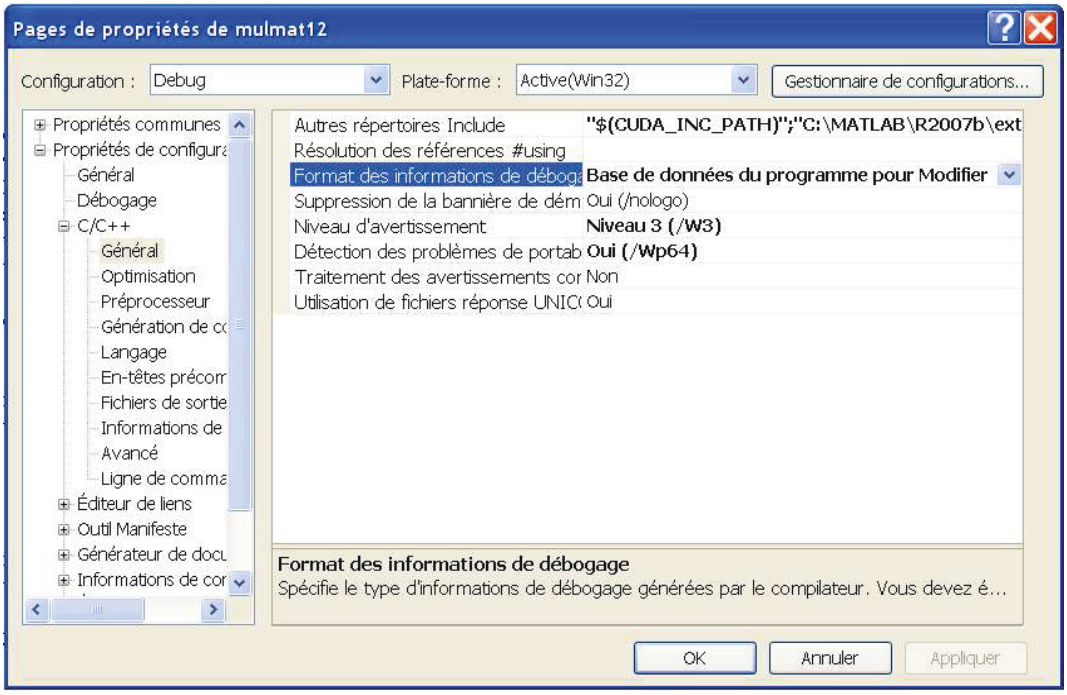
Dans cette fenêtre de propriétés, aller dans les menus suivants : *Propriétés de configuration* → *Général* puis sélectionner *Bibliothèque dynamique (.dll)* dans le champ *Type de configuration* puis *Appliquer*.


mulmat12 - Visual C++ 2005 Express
Fichier Edition Affichage Projet Générer Déboguer
Précédent
Comment faire
Explorateur de solutions...
mulmat12.cu
Solution 'mulmat12' (1 projet)
mulmat12
Fichiers
Générer
Régénérer
Nettoyer
Projet uniquement
Règles de génération personnalisée...
Ordre de génération des outils...
Ajouter
Références...
Ajouter une référence Web...
Définir comme projet de démarrage
Déboguer
Couper
Coller
Supprimer
Renommer
Décharger le projet
Propriétés


Pages de propriétés de mulmat12
Configuration : Toutes les configurations
Plate-forme : Active(Win32)
Gestionnaire de configurations...
Propriétés communes
Propriétés de configuration
Général
Débogage
C/C++
Générateur de bibliothè
Générateur de docume
Informations de consul
Événements de généra
Étape de génération pe
Général
Répertoire de sortie $(SolutionDir)$(ConfigurationName)
Répertoire intermédiaire $(ConfigurationName)
Extensions à supprimer lors du nett *.obj;*.ilk;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.pgd;$(Tar
Fichier journal de génération $(IntDir)\BuildLog.htm
Feuilles de propriétés de projet héri
Paramètres par défaut du projet
Type de configuration Bibliothèque dynamique (.dll)
Utilisation des MFC Utiliser les bibliothèques Windows standard
Utilisation des ATL N'utilisant pas ATL
Réduction de l'utilisation des CRT d Non
Jeu de caractères Utiliser le jeu de caractères Unicode
Prise en charge du Common Langu Pas de prise en charge du Common Language Runtime
Optimisation de l'ensemble du progi
Type de configuration
Spécifie le type de sortie généré par cette configuration.
OK
Annuler
Appliquer

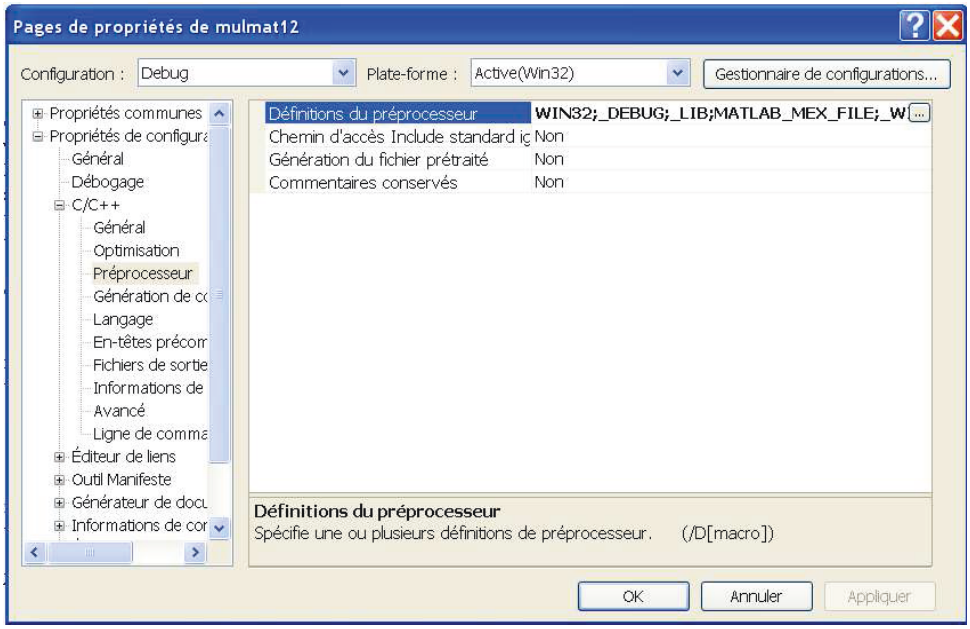
Ensuite :

- Aller ensuite dans les menus et sous-menus suivants : *Propriétés de configuration* → *C/C++* → *Général* et entrer la ligne *C:\MATLAB\R2007b\extern\include;$(CUDA_INC_PATH)* dans le champ *Autres répertoires include*.
- Cliquer sur *Appliquer*.

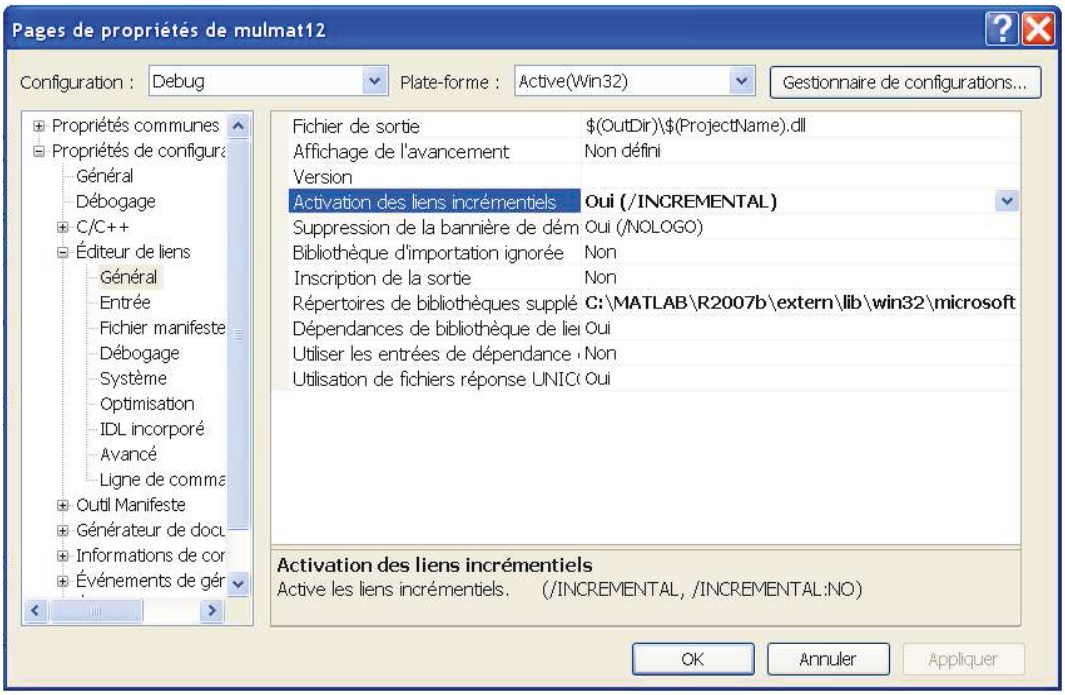



- Aller ensuite dans le menu de configuration du préprocesseur : *Propriétés de configuration* → *C/C++* → *Préprocesseur* et ajouter (aux options déjà présentes) dans le champ *Définitions du préprocesseur* :
  - mode *Debug* : *MATLAB_MEX_FILE;_WINDOWS;MULMAT_EXPORTS*
  - mode *Release* : *MATLAB_MEX_FILE;_WINDOWS;MULMAT_EXPORTS*

Si nécessaire remplacer (dans les deux modes) le nom *MULMAT* par un nom plus représentatif du projet par exemple *CFFT* dans le cas d'un programme de FFT.




- Clique sur *Appliquer*.
- Aller dans le menu de l'éditeur de lien : *Propriétés de configuration → Editeur de liens → Général* et sélectionner (en mode *DEBUG*) dans le champ *Activation des liens incrémentiels : Oui (/INCREMENTAL)*




- Dans le même menu, entrer la ligne *C:\MATLAB\R2007b\extern\lib\win32\microsoft; $(CUDA_LIB_PATH)* dans le champ *Répertoires de bibliothèques supplémentaires*.
- Dans le menu *Entrée* des propriétés de l'éditeur de lien, taper dans le champ *Dépendances supplémentaires* :
  a) pour un projet *MEX-CUDA* :
     - mode *Debug* : *cudart.lib cuda.lib cutil32D.lib libmat.lib libmex.lib libmx.lib*
     - mode *Release* : *cudart.lib cuda.lib cutil32.lib libmat.lib libmex.lib libmx.lib*

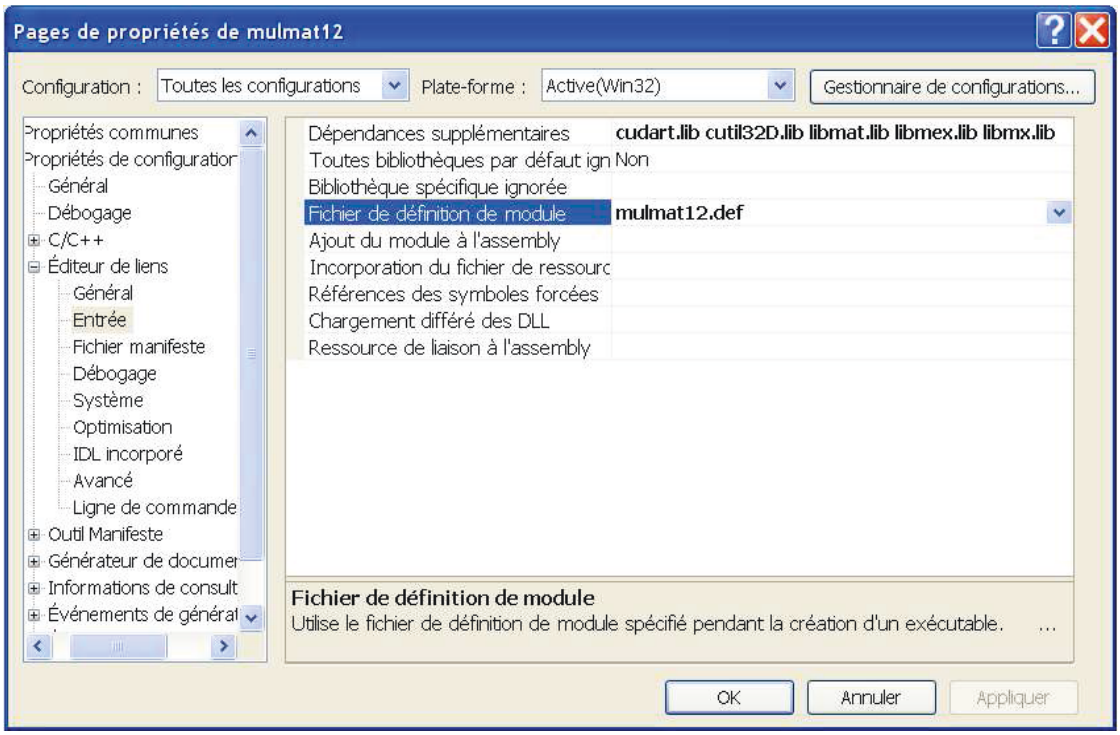
  b) pour un projet *MEX-CUBLAS* :

- mode *Debug* : *cudart.lib cublas.lib cutil32D.lib libmat.lib libmex.lib libmx.lib*
- mode *Release* : *cudart.lib cublas.lib cutil32.lib libmat.lib libmex.lib libmx.lib*

c) pour un projet *MEX-CUFFT* :

- mode *Debug* : *cudart.lib cufft.lib cutil32D.lib libmat.lib libmex.lib libmx.lib*
- mode *Release* : *cudart.lib cufft.lib cutil32.lib libmat.lib libmex.lib libmx.lib*

- Dans le même menu, taper *mulmat12.def* dans le champ *Fichier de définition de module* (mode *Toutes les configurations*) et remplacer *mulmat12* par le nom du programme si nécessaire.




- Enregistrer cette configuration en cliquant sur *Appliquer*.

## 5.2 Description détaillée de la procédure de configuration du projet :

### 5.2.1 Dans le menu principal

Le choix du type de configuration en *Bibliothèque dynamique (.dll)* permet de préciser à Visual C++ de créer une bibliothèque dynamique (à partir du programme CUDA) qui pourra être appelée par Matlab.

Dans le sous-menu *C/C++* → *Général*, doivent être listés les répertoires *include* utiles au programme dans le champ *Autres répertoires include* :

- répertoire *include* de Matlab : *C:\MATLAB\R2007b\extern\include* (dans lequel se trouve *mex.h*),
- répertoire *include* de CUDA™ : *CUDA_INC_PATH* (dans lequel se trouve *cuda.h*).

Le terme *CUDA_INC_PATH* est une variable d'environnement correspondant à un chemin donné à Visual C++ lui indiquant l'emplacement des fichiers d'en-têtes. Ce terme correspond à un chemin se terminant par : *\inc* ou *\include* c'est à dire dans le cadre de ce projet : *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\include*. Le chemin exact contenu dans *CUDA_INC_PATH* se trouve en suivant la procédure décrite dans la configuration de fichier au sujet des variables d'environnement (Annexe 4 §4.3).

### 5.2.2 Dans les propriétés de configuration du préprocesseur

Le symbole *_DEBUG* est une macro permettant d'activer les fonctionnalités de débogage à la compilation. Ce symbole indique à l'éditeur de liens d'inclure les informations de débogage, qui auront le format spécifié par l'un des termes suivants : */Z7*, */Zd*, */Zi* ou */ZI*.

*NDEBUG* est composé de deux parties :

- *N*
- *DEBUG*

*N* a pour but d'afficher les commandes sans les exécuter ; les commandes de prétraitement sont exécutées.

Le terme *DEBUG* est une option qui crée des informations de débogage pour le fichier *.exe* ou la *.dll*.

Le terme *MATLAB_MEX_FILE* est une définition de préprocesseur qui rend possible la compilation et la génération de fichiers *MEX* avec l'environnement de développement intégré (*IDE*) de Microsoft Visual C++.

Le terme *_WINDOWS* est une variable système incluse pour la compatibilité rétroactive et contient toujours la valeur vraie en cas d'utilisation de Visual FoxPro.

Le terme *EXPORTS* permet de présenter une section d'une définition ou plus. Chaque définition est sur une ligne séparée. Le projet *dll* définit *MULMAT_EXPORTS* lors de la generation. *MULMAT_EXPORTS* est défini pour des fichiers sources générés pour une bibliothèque partagée.

### 5.2.3 Dans les propriétés de configuration de l'éditeur de liens

Le terme */INCREMENTAL* contrôle la façon dont l'éditeur de liens gère les liaisons incrémentielles.

Le terme */INCREMENTAL* est implicite lorsque */DEBUG* est spécifié.

Un programme lié de façon incrémentielle équivaut en termes de fonctionnalités, à un programme lié de façon non incrémentielle. Toutefois, comme ils sont préparés pour des liens incrémentiels à venir, les fichiers exécutables liés de façon incrémentielle (*.exe*) ou les bibliothèques de liens dynamiques (*dll*) présentent les caractéristiques suivantes :

- Ils sont plus volumineux qu'un programme lié de façon non incrémentielle du fait du remplissage du code et des données. Le remplissage permet à l'éditeur d'augmenter la taille des fonctions et des données sans recréer le fichier *.exe* .
- Ils peuvent contenir des *thunks* de branchement gérant le réadressage des fonctions vers les nouvelles adresses.

Pour s'assurer que la génération de la version *release* ne contient pas de remplissages ou de thunks, il faut lier le programme de façon non incrémentielle.

- Dans le répertoire *C:\MATLAB\R2007b\extern\lib\win32\microsoft*, l'éditeur de liens trouvera les bibliothèques suivantes : *libmat.lib*, *libmex.lib* et *libmx.lib* (bibliothèques nécessaires aux parties *MEX* du programme pour que le programme puisse être appelé par Matlab).
- Dans le répertoire *$(CUDA_LIB_PATH)*, l'éditeur de liens trouvera la bibliothèque *cudart.lib* (bibliothèque runtime de CUDA™). Le terme *CUDA_LIB_PATH* est une variable d'environnement correspondant à un chemin donné à Visual C++ lui

indiquant l'emplacement des bibliothèques *.lib*. Ce terme correspond à un chemin se terminant par : *\lib* ou contenant *\lib* c'est à dire dans le cadre de ce projet : *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\lib\Win32*. Le chemin exact contenu dans *CUDA_LIB_PATH* se trouve en suivant la procédure décrite dans la configuration de fichier au sujet des variables d'environnement (Annexe 4 §4.3).

Les bibliothèques *.lib* sont les bibliothèques dont l'éditeur de liens a besoin pour faire l'opération de liaison du programme. L'éditeur cherchera ces bibliothèques dans les répertoires précédemment listés.

Le fichier de définition de module *mulmat12.def* fournit à l'éditeur de liens des informations sur les exportations et les attributs ainsi que d'autres données concernant le programme devant être lié. Ce fichier est utile lors de la génération de la *dll* du programme. Son contenu est décrit dans l'Annexe 4 §2).

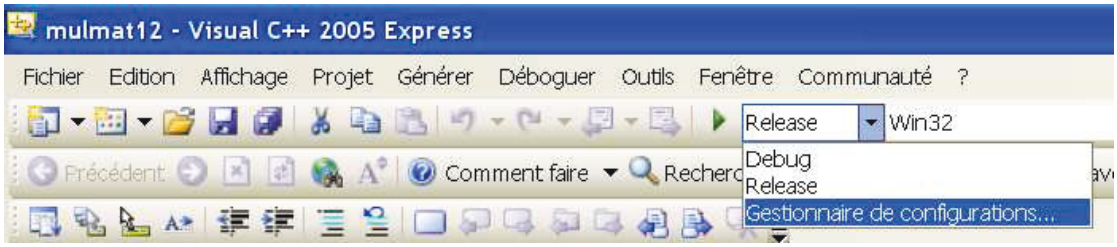
# 6. Création et configuration d'un mode émulation pour projet MEX-CUDA, MEX-CUBLAS ou MEX-CUFFT

## 6.1 Description générale

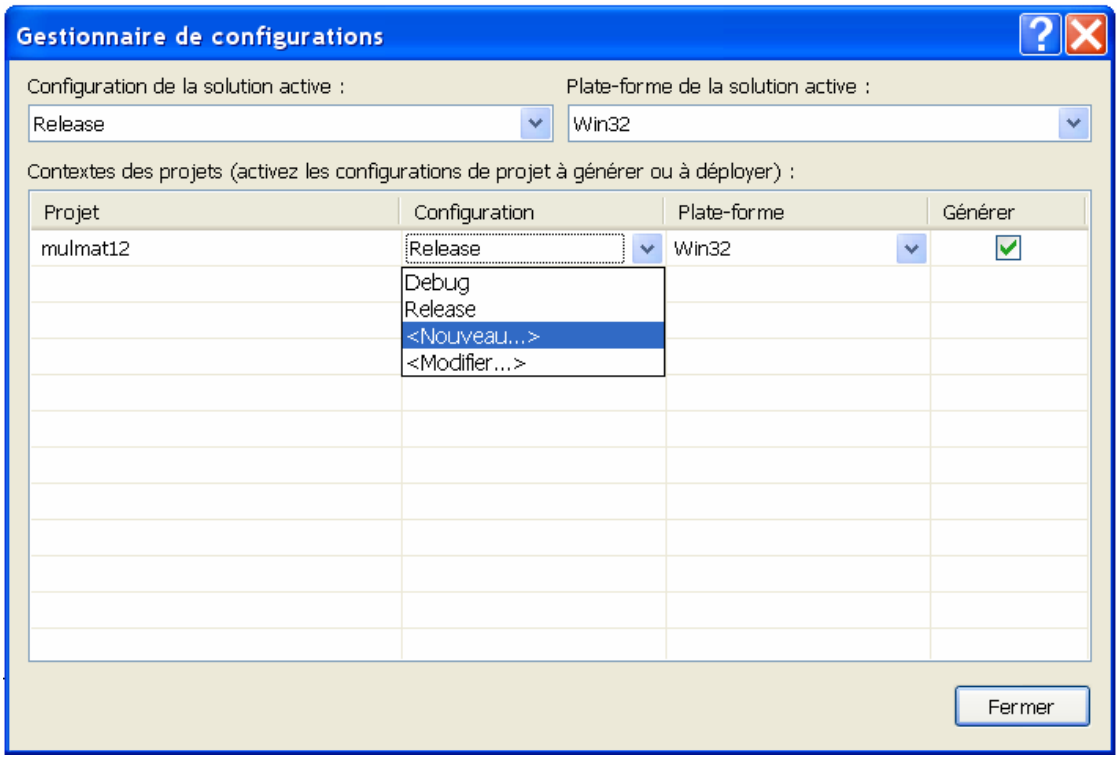
Lorsqu'on veut tester un programme mais qu'on n'a pas de carte graphique sur laquelle le tester, il est possible de tester le bon fonctionnement du programme sur CPU (bien qu'il sera évidemment plus lent) en l'exécutant dans un mode d'émulation. Le mode d'émulation n'existant pas par défaut, il devra être créé sous deux formes particulières :

- *EmuDebug* pour l'émulation en mode *Debug*
- *EmuRelease* pour l'émulation en mode *Release*

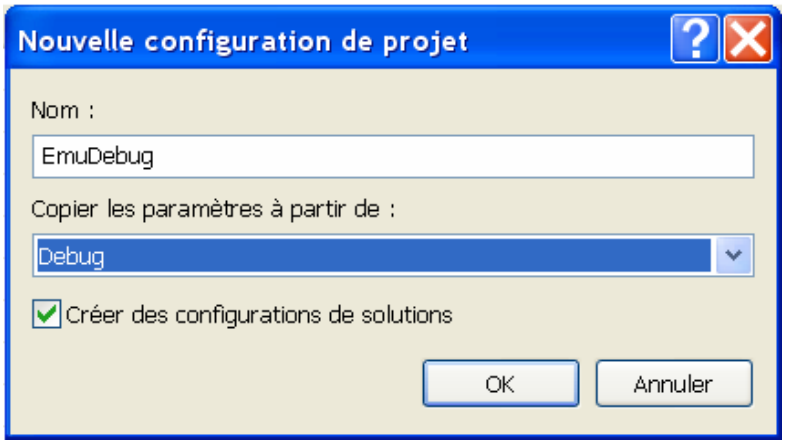
Dans Visual C++ 2005 Express, cliquer sur la « flèche vers le bas » située à droite de la configuration courante.



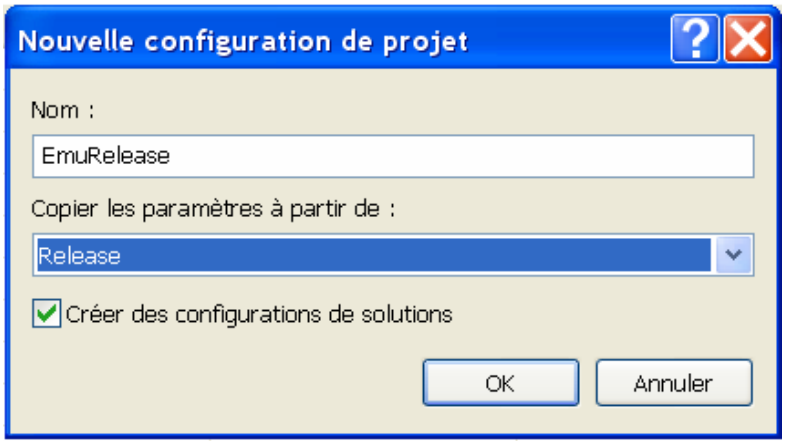

Dans la colonne *Configuration*, cliquer sur *Nouveau*.




Entrer *EmuDebug* dans le champ *Nom* de la fenêtre *Nouvelle configuration de projet*. Choisir ensuite *Debug* dans le champ *Copier les paramètres à partir de :*




Faire la même chose pour la configuration *EmuRelease* :




Ensuite cliquer sur *OK*, puis sur l'icône *Fermer* du *Gestionnaire de configurations*.

Une fois les modes *EmuDebug* et *EmuRelease* créés, il faut configurer celles de leurs options qui diffèrent des modes (respectivement) *Debug* et *Release*.

Cette légère modification de la configuration se fait par les étapes suivantes :

- Aller dans les menus et sous-menus suivants dans les propriétés du fichier : *Propriétés de configuration → Etape de génération personnalisée → Général* puis dans le champ *Ligne de commande* :
  - Pour un projet *CUDA* en mode *EmuDebug* et *EmuRelease* : ajouter *-deviceemu* entre *-ccbin "$(VCInstallDir)bin"* et *–c*.
  - Pour un projet *CUBLAS* en mode *EmuDebug* et *EmuRelease* : ajouter *-lcublasemu* entre *-ccbin "$(VCInstallDir)bin"* et *–c*.
  - Pour un projet *CUFFT* en mode *EmuDebug* et *EmuRelease* : ajouter *-lcufftemu* entre *-ccbin "$(VCInstallDir)bin"* et *–c*.
  - Remplacer */O2* par */Od* pour le mode *EmuRelease* uniquement et quel que soit le type de projet (*CUDA*, *CUBLAS* ou *CUFFT*).
- Aller ensuite dans les propriétés du projet et :
  - Dans les menus et sous-menus *Propriétés de configuration → C/C++ → Optimisation*, remplacer *Augmenter la vitesse (/O2)* par *Désactivé (/Od)* pour le mode *EmuRelease* uniquement dans le champ *Optimisation*.
  - Dans les menus et sous-menus *Editeur de liens → Entrée → dépendances supplémentaires*, pour les modes *EmuDebug* et *EmuRelease* uniquement, remplacer les bibliothèques :
    - *cublas.lib* par *cublasemu.lib*
    - *cufft.lib* par *cufftemu.lib*

## 6.2 Description détaillée de la création et configuration des deux modes d'émulation

Le fait de choisir *Debug* dans *Copier les paramètres à partir de :* signifie que Visual C++ va copier le contenu de la configuration *Debug* pour la mettre dans la configuration *EmuDebug*. Ceci permet de gagner du temps en évitant de refaire une longue configuration (contenant de nombreux paramètres identiques) puisqu'il suffira ensuite de modifier uniquement les paramètres qui diffèrent réellement entre *Debug* et *EmuDebug*.

La même remarque est valable pour la configuration de *EmuRelease* copiée à partir de la configuration de *Release*.

Le terme *-deviceemu* est une option du compilateur *NVCC* qui génère du code pour la bibliothèque d'émulation GPGPU. Cette option est particulièrement utile lorsqu'on n'a pas de carte graphique compatible CUDA mais qu'on veut quand même essayer d'exécuter un programme. Cette option fait le lien avec un émulateur de carte graphique CUDA qui s'exécute sur l'hôte (le CPU). L'émulateur devient la cible pour tous les appels CUDA et exécute le kernel. Le programme s'exécutera de la même manière mais plus lentement que si une carte graphique était présente.

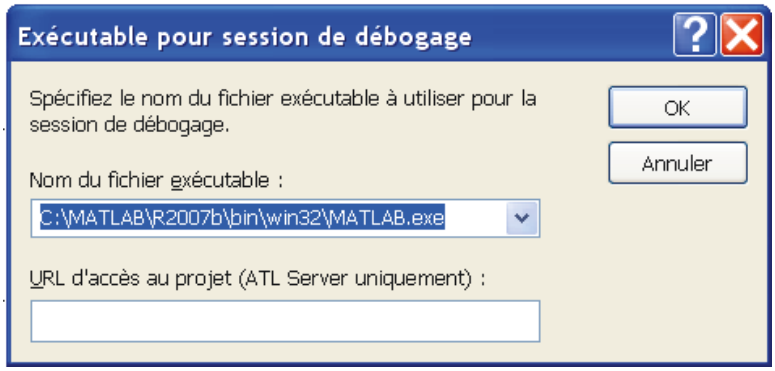
Ce projet a été entièrement basé sur les instructions de l'API du moteur d'exécution (*runtime*) CUDA. Il existe un autre jeu d'instruction qui s'utilise par l'API du pilote CUDA. Le mode émulation n'existe pas lorsqu'on utilise l'API du pilote CUDA.

# 7. Configuration de MATLAB pour l'exécution d'un projet MEX-CUDA, MEX-CUBLAS, MEX-CUFFT ou autre)

Une fois le projet et le fichier code configurés, si le code est prêt, il ne reste plus qu'à le compiler, générer la solution et l'exécuter. Le plus rapide est d'utiliser les raccourcis clavier :

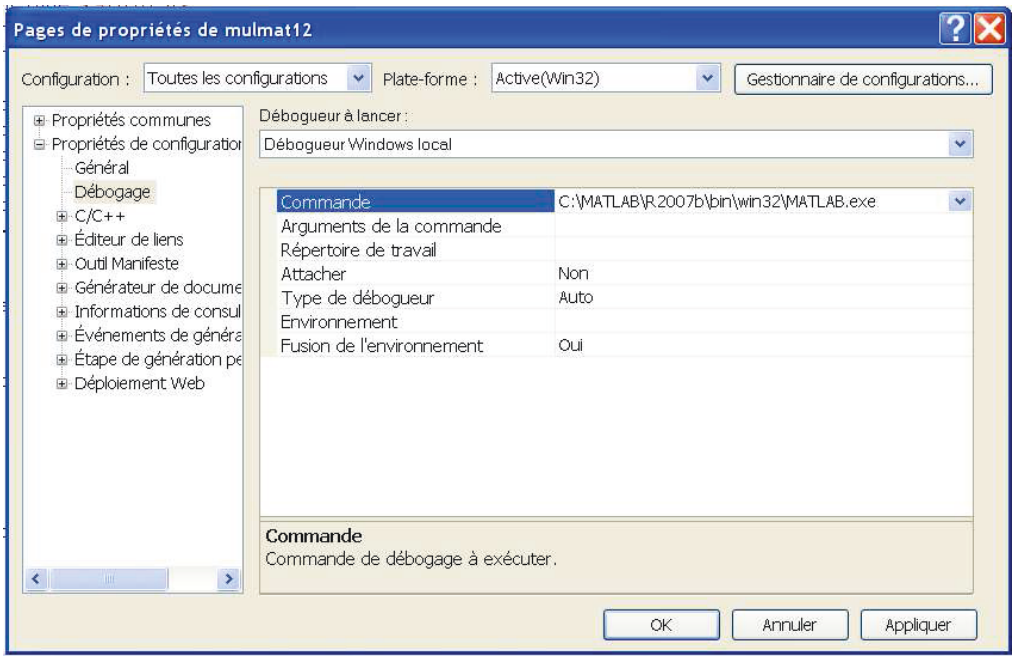
- compiler : Ctrl+F7
- générer : F7
- exécuter : Ctrl+F5

Lors de la première exécution d'un nouveau projet dans Visual C++ 2005, la fenêtre suivante s'ouvre :




Dans cette fenêtre, donner le chemin d'un fichier exécutable, c'est-à-dire MATLAB dans ce cas. Lors de toutes les exécutions de ce programme dans la même configuration, le chemin restera enregistré et n'aura donc pas besoin d'être de nouveau mentionné.

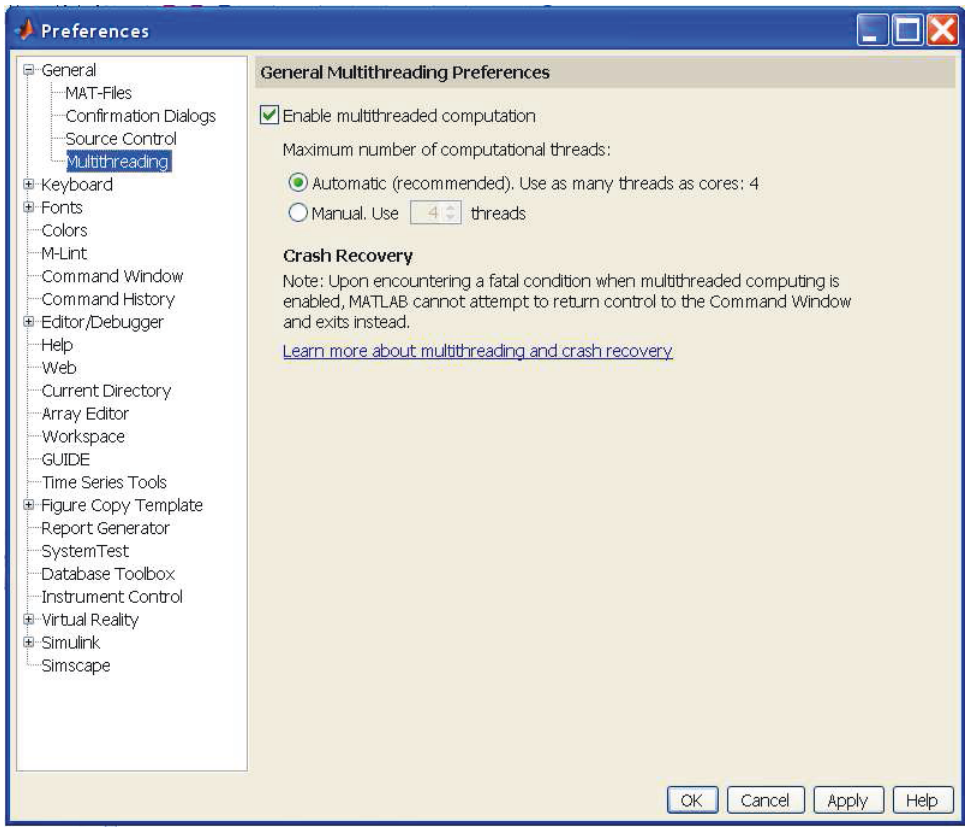
Lors d'exécutions ultérieures, ce message n'apparaîtra plus. S'il est nécessaire de changer le chemin du fichier exécutable (donc de MATLAB), il est encore possible de le faire en allant dans les propriétés du projet *mulmat12* et en cliquant sur débogage dans le menu *Propriétés de configuration*. Il suffit ensuite de spécifier le chemin correct dans lequel se trouve *MATLAB.exe*. Attention à bien choisir *Toutes les configurations* (*Debug*, *EmuDebug*, etc.) dans *Configuration* avant de cliquer sur *Appliquer*.




Ensuite, si on veut comparer les performances d'un programme CUDA avec les performances maximales du CPU, on peut choisir de configurer MATLAB de façon à ce

qu'il s'exécute sur plusieurs cœurs du CPU. Le CPU utilisé dans le cadre de ce mémoire comporte 4 cœurs, donc pour configurer MATLAB, aller dans les menus et sous-menus suivants de MATLAB : *File* → *Preferences* → *General* → *Multithreading* et cocher :

- *Enable multithreaded computation* pour autoriser le calcul multitâche
- puis :
  - *Automatic* ou
  - *Manual* et *4 threads*



# Annexe 5
# Outils de développement CUDA

Dans le kit de développement CUDA (*CUDA Toolkit*), il existe d'autres outils et bibliothèques qui n'ont pas été utilisés dans ce projet. Le but de cette annexe est donc de présenter rapidement ces fonctionnalités supplémentaires.

## 1. Autres bibliothèques disponibles

Il existe de nombreuses autres bibliothèques. Les plus utiles sont :

- La bibliothèque CURAND pour générer des nombres aléatoires,
- La bibliothèque CUSPARSE pour gérer les matrices creuses.

## 2. Procédure de débogage d'un programme MEX-CUDA multitâche sous Visual C++

Pour déboguer un programme multitâche, suivre la procédure suivante :

- Sélectionner le mode *Debug* dans la barre d'outils de Visual C++.
- Poser des points d'arrêts devant chaque instruction avant lesquelles on souhaite s'arrêter avec *Basculer le point d'arrêt* dans la rubrique *Déboguer* du menu de Visual C++ ou bien plus simplement en appuyant sur la touche F9 (raccourci clavier).
- Cliquer sur *Exécuter sans débogage* dans la rubrique *Déboguer* du menu de Visual C++ ou bien, plus simplement en appuyant sur les touches Ctrl+F5.
- Lorsque le curseur de la fenêtre de commande de MATLAB apparaît, retourner sous Visual C++ et cliquer sur *Attacher au processus* dans la rubrique *Outils* ou dans la rubrique *Déboguer* du menu de Visual C++.
- Cliquer sur l'application *MATLAB.exe* dans la liste de processus qui apparaît, puis sur *Attacher*.
- Retourner sous MATLAB et double-cliquer sur le répertoire *Debug* dans la fenêtre *Répertoire courant* (Current Directory).
- Lancer le programme MATLAB dans lequel le programme MEX-CUDA est appelé.
- Visual C++ réapparaît alors automatiquement avec une flèche sur le premier point d'arrêt. Le programme a été exécuté jusqu'à ce point d'arrêt.
- Pour passer au point d'arrêt suivant, cliquer sur *Continuer* dans la rubrique *Déboguer* du menu de Visual C++, ou bien plus simplement appuyer sur la touche F5.
- Pour faire du pas à pas entre les points d'arrêt, cliquer sur *Pas à pas principal* dans la rubrique *Déboguer* du menu de Visual C++, ou bien plus simplement appuyer sur la touche F10.
- Dans la fenêtre du bas à gauche (toujours dans Visual C++), on peut voir une fenêtre dans laquelle est affichée selon l'onglet sélectionné :
  - la valeur courante des variables automatiques
  - la valeur courante des variables locales
  - l'état des tâches courantes sous l'onglet *Threads*
  - la liste des bibliothèques dynamiques (*dll*) utilisées au point d'arrêt sous l'onglet *Modules*
- Une fois le débogage terminé, cliquer sur *Arrêter le débogage* (visible seulement au moment du débogage) dans la rubrique *Déboguer* du menu de Visual C++ ou bien, plus simplement en appuyant sur la touche Maj+F5.

# 3. Autres outils de développement

## 3.1. Calculateur de taux d'occupation (CUDA occupancy calculator)

### 3.1.1 Présentation

Le calculateur de taux d'occupation (CUDA Occupancy Calculator) est un tableau Excel fourni avec le *CUDA Toolkit*. Il se trouve donc dans le répertoire du *CUDA Toolkit*. S'il n'y est pas présent, la dernière version peut être téléchargée sur le site de NVIDIA. Vérifier que la version du calculateur utilisé est suffisamment récente, les versions anciennes ne prenant pas en compte les capacités de calcul de 2.0 et supérieures. La plus récente disponible en date du 29 juillet 2011 est la version 2.4. La version est visible sur la ligne 50 de l'onglet *Calculator* de la feuille de calcul.

Ce calculateur peut être utilisé pour déterminer si, lors du lancement d'un kernel, le nombre de tâches par bloc est optimal ou non. Son contenu n'est disponible qu'en langue anglaise.

Ce tableau Excel comporte 4 pages accessibles directement :

- soit par les onglets situés en bas de feuille (*Calculator*, *Help*, *GPU Data* et *Copyright & License*).
- soit en cliquant sur les liens hypertextes (qui renvoient aux onglets concernés).

Ce tableau permet obtenir, à partir de 5 données de base, de nombreuses informations sur les paramètres générés par une configuration particulière d'un projet CUDA.

Les données à entrer sont :

- La version de processeur (*Compute Capability*)
- La configuration de la taille de la mémoire partagée en octets (*Shared Memory Size Config*)
- Le nombre de tâches par blocs (*Threads Per Block*)
- Le nombre de registres par tâches (*Registers per Thread*)
- La quantité de mémoire partagée par bloc (en octets) (*Shared Memory Per Block*)

### 3.1.2 Recherche de la capacité de calcul

La capacité de calcul dépend du type de processeur graphique présent sur la carte graphique. Selon le modèle de carte graphique utilisée, la version de processeur (*compute capability*) se trouve dans l'annexe A du guide de programmation de CUDA™. Si on ne connaît pas le modèle de carte graphique présent dans le PC utilisé, il est possible de le trouver :

- Soit en cliquant droit sur le *Bureau* puis en allant dans *Panneau de configuration NVIDIA → Informations Système → Affichage → Elements*.
- Soit en allant dans le menu *démarrer → Panneau de configuration → Panneau de configuration NVIDIA → Informations Système → Affichage → Elements*.
- Soit en éxécutant le programme CUDA après avoir ajouté la fonction *cudaGetDeviceProperties*.

### 3.1.3 Recherche de la configuration de la taille de la mémoire partagée

Dans les précédentes générations d'architecture CUDA (versions 1.0, 1.1, 1.2, 1.3), la taille de mémoire partagée était fixée à 16 ko par multiprocesseur. Sur les architectures Fermi, une partie de la mémoire partagée est partagée avec une partie de la

mémoire cache L1 mais la quantité maximale de mémoire partagée est de 48 ko. La répartition de mémoire est configurable (par le programmeur) de la manière suivante :

- soit 48 ko de mémoire partagée et 16 ko de mémoire cache L1,
- soit 16 ko de mémoire partagée et 48 ko de mémoire cache L1.

### 3.1.4 Recherche du nombre de tâches par bloc

En général, le nombre de tâches par bloc est connu du programmeur puisqu'il a choisi lui-même les dimensions des blocs.

Dans les rares cas où le programmeur ne connaîtrait pas le nombre de tâches par bloc, il peut les connaître en exécutant son programme après avoir ajouté une instruction d'extraction de la valeur de la structure prédéfinie `blockDim`. Une fois connues les dimensions de bloc, il suffit ensuite de les multiplier entre elles pour obtenir le nombre de tâches par bloc.

### 3.1.5 Recherche du nombre de registres par tâche et de la quantité de mémoire partagée

Pour déterminer le nombre de registres utilisés par tâche dans le kernel ou bien la quantité de mémoire partagée réellement utilisée, il suffit de compiler le code du kernel en utilisant l'option *--ptxas-options=-v* dans la ligne de commande de la configuration du fichier CUDA.

Dans l'instruction *--ptxas-options=-v* , la partie *--ptxas-options* est une instruction de ligne de commande permettant de spécifier des options directement à l'assembleur d'optimisation *ptx*. La partie *-v* (mode *verbose*) de l'instruction ci-dessus est une option qui permet de lister les commandes de compilation générées par le pilote de compilateur, mais ne supprime pas leur exécution. Lorsque cette option est présente dans la ligne de commande du compilateur, deux lignes supplémentaires apparaissent dans la fenêtre de sortie de Visual C++ du type :

```
1>ptxas info    : Compiling entry function '_Z12MatMulKernel6MatrixS_S_' for 'sm_20'
1>ptxas info    : Used 20 registers, 4096+0 bytes smem, 80 bytes cmem[0]
```

On y trouve en particulier le nombre de registres réellement utilisés (`20 registers`) et la quantité de mémoire partagée réellement utilisée (`4096+0 bytes smem`).

Pour les anciennes versions de CUDA™, il est également possible d'obtenir la quantité de mémoire partagée réellement utilisée et le nombre de registres utilisés en ouvrant le fichier *.obj* du programme concerné dans un éditeur de texte simple du type WordPad (non lisible sous Word). Parmi les nombreuses lignes se trouve un paragraphe du type :

```
___architecture {sm_10}
abiversion   {1}
modname      {cubin}
code {
       name = _Z15square_elementsPfS_i
       lmem = 0
       smem = 28
       reg  = 2
       bar  = 0
       bincode {
              0x10004205 0x0023c780 0xa0000005 0x04000780
              0x60014c01 0x00204780 0x3000cdfd 0x6c20c7c8
              0x30000003 0x00000280 0x30020005 0xc4100780
              0x2000c801 0x04204780 0xd00e0001 0x80c00780
              0x2101ea04 0xc0000000 0xd00e0201 0xa0c00781
```

```
        }
}
```

Dans ce paragraphe, les données les plus utiles sont :

- L'architecture pour laquelle a été compilé le programme est identifiée par le terme `10` dans le terme `sm_10`. L'architecture 10 correspond à la version de processeur `1.0`. De manière générale une architecture *xy* correspond à une version de processeur *x.y*.
- La quantité de mémoire partagée identifiée par le terme `smem`.
- Le nombre de registres utilisés identifié par le terme `reg`.

La quantité de mémoire partagée utilisée et la quantité de registres utilisés peuvent également être trouvés avec le *Compute Visual Profiler*.

Attention, lorsqu'on ferme le calculateur de taux d'occupation, il est systématiquement demandé si on souhaite sauvegarder les changements, cliquer sur *Non* sauf si on veut sauvegarder un certain profil de configuration pour une raison particulière.

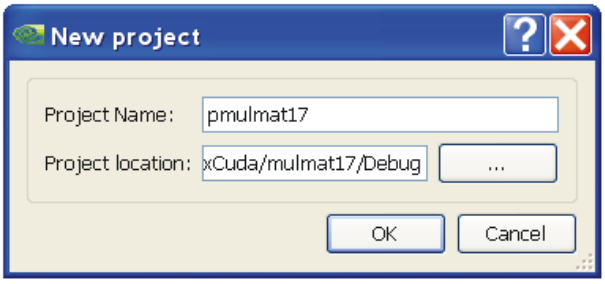
## 3.2. Compute Visual Profiler

Le *Compute Visual Profiler* est fourni avec le *CUDA Toolkit* dans le répertoire *computeprof*.

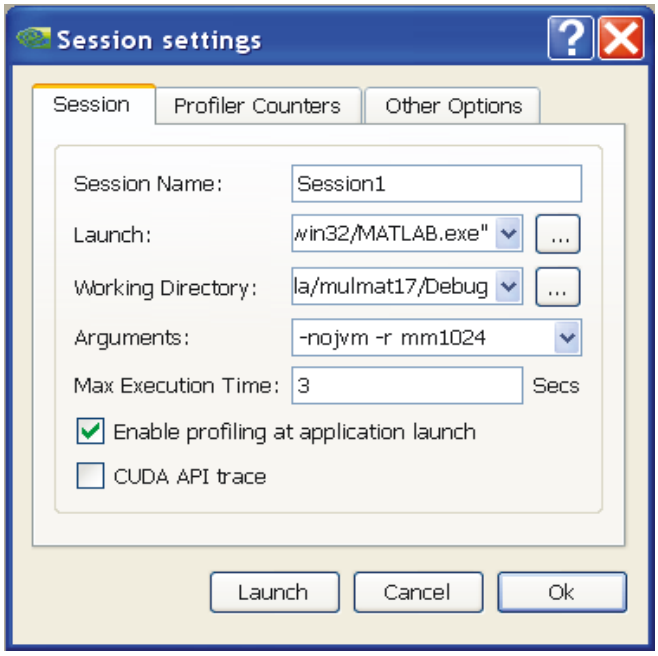
Le *Compute Visual Profiler* est un outil de profilage basé sur une interface utilisateur graphique et qui peut être utilisé pour mesurer la performance d'un programme CUDA. Cette application réalise des mesures et les affiche sous forme de tableaux ou de graphiques.

Pour faire le profilage d'un programme, suivre préalablement les étapes suivantes de configuration du *Compute Visual Profiler* :

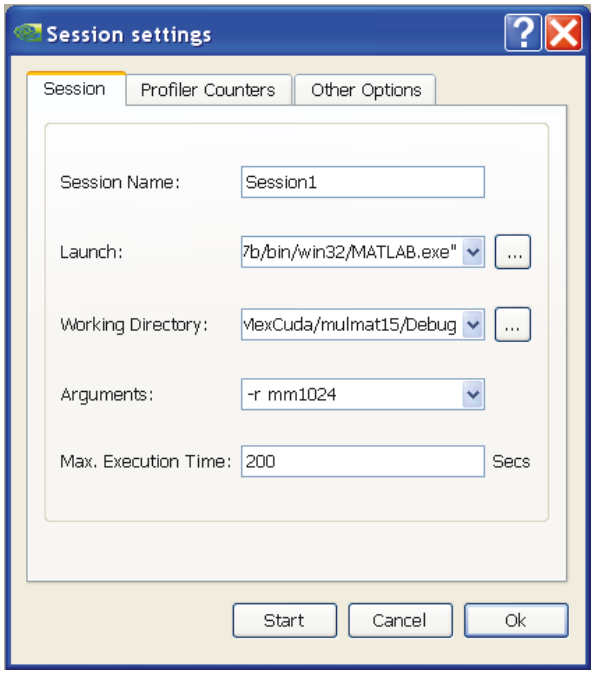
- Aller dans MATLAB et ajouter `Quit` à la fin du programme Matlab.
- Aller dans Visual C++ et ajouter `cudaThreadExit();` à la fin du programme CUDA-C si ce n'est pas déjà fait.
- Lancer *Compute Visual Profiler* :
  - soit en allant dans *démarrer → Tous les programmes → NVIDIA Corporation → CUDA Toolkit → v3.2 → Compute Visual Profiler → Compute Visual Profiler*
  - soit en passant par le poste de travail dans un répertoire nommé *computeprof* : *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\computeprof\bin*
- Attention, plusieurs versions de Profiler existent. Il est conseillé d'utiliser la dernière version. Pour trouver la version, il suffit de cliquer sur *About Compute Visual Profiler* dans le menu *Help* du *Compute Visual Profiler*.
- Dans le *Compute Visual Profiler*, cliquer sur *New* dans le menu *File*.
- Donner un nom au projet de profilage (nom différent du nom utilisé dans Visual C++) dans *Project Name*.
- Entrer le chemin du répertoire dans lequel les fichiers du projet seront sauvegardés dans *Project Location*.




- Cliquer sur *OK*.
- Dans la fenêtre *Session settings* qui apparaît, aller dans l'onglet *Session*.
- Dans *Session Name*, le chiffre présent dans le nom de la session est incrémenté par défaut à chaque nouvelle session.
- Entrer dans *Launch* le nom du logiciel à profiler et son chemin : *"C:/MATLAB/R2007b/bin/win32/MATLAB.exe"*.
- Entrer dans *Working Directory* le chemin du répertoire de travail à utiliser pour exécuter le logiciel spécifié dans *Launch* : *C:/users/paoletti/Projet/EssaisMexCuda/mulmat17/Debug*.
- Entrer dans *Arguments* les arguments de ligne de commande à passer au logiciel spécifié dans *Launch* : *-r mm1024* .
- Ajouter *-nojvm* dans *Arguments* pour récupérer les résultats.
- Entrer dans *Max. Execution Time* la durée maximale pour laquelle *Compute Visual Profiler* attend la fin d'exécution du logiciel spécifié dans *Launch*, c'est-à-dire : 3 Secs. Après cette durée limite, l'exécution du logiciel est abandonnée.
- Laisser la case *Enable profiling at application launch* cochée, car sinon le profilage ne serait pas exécuté lors de l'exécution du logiciel spécifié dans *Launch*.
- Dans les autres onglets, les options cochées représentent tous les paramètres qui vont être mesurés lors de l'exécution du programme MEX-CUDA sous MATLAB. Cocher ou décocher les options selon les besoins de mesure. Le nombre d'exécutions que le profiler réalisera est proportionnel à la quantité d'options cochées.
- Cliquer sur *Launch*.
- Si on clique sur *Ok* à la place de *Launch*, les informations entrées sont enregistrées et il est possible de les retrouver en cliquant sur l'icône *Session Settings* ou en allant dans le menu *Session* → *Settings*.



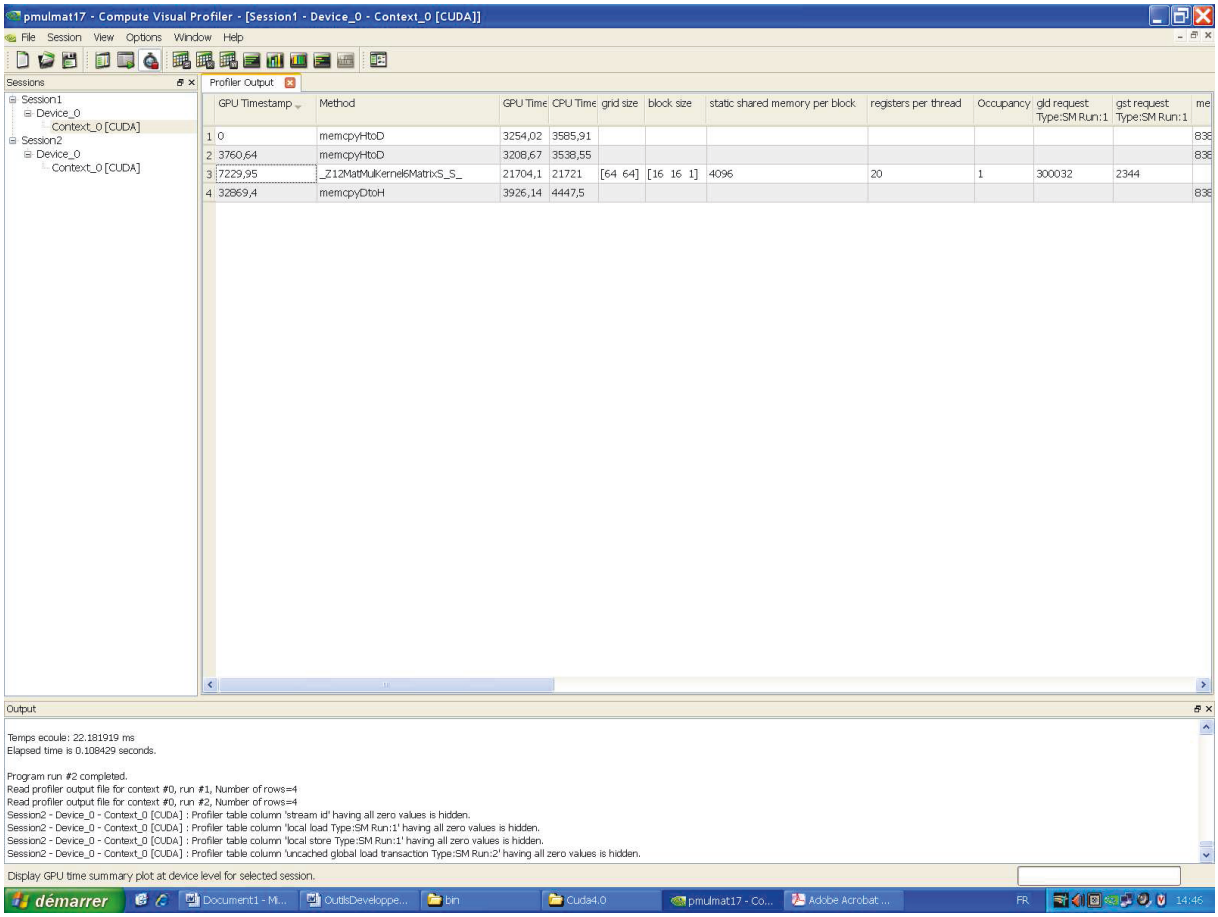

Dans le cas ou on utilise une des premières versions de CUDA™, la fenêtre *Session Settings* apparaîtra comme dans la figure ci-dessous.




Une fois le profilage exécuté, on peut y trouver de nombreuses informations utiles, en particulier :

- La taille des blocs (en `x`, `y` et `z`).
- La taille de la grille
  - en `x`, `y` et `z` pour les versions les plus récentes de CUDA™ ou
  - `x` et `y` seulement pour les versions les moins récentes).
- La quantité de mémoire partagée utilisée.
- Le nombre de registres utilisés par tâche.
- Les temps d'exécution du kernel (en µs) et de chaque transfert mémoire. Le profiler permet ainsi d'éviter d'avoir recours à une série d'instructions CUDA fastidieuses pour mesurer les temps des 3 transferts mémoire et le temps d'exécution du kernel.

- Une fois le profilage achevé, on a accès à plusieurs types d'informations :
  - Des informations de temps en cliquant sur les icônes *GPU time summary plot*, *GPU time height plot* et *GPU time width plot* qui offrent des réprésentations du même temps de calcul et des mêmes temps de transfert de données.
  - Des informations résumées en cliquant sur l'icône *Summary table*.
  - Les principales données concernant le kernel seul en cliquant sur l'icône *Kernel table*.
  - Les principales données concernant les transferts mémoires CPU/GPU seuls en cliquant sur l'icône *Memcopy Table*.




- Il est possible de sauver les données créées par le profilage en cliquant sur l'icône *Save* ou en cliquant sur *Save* dans le menu *File*. Conserver les données de profilage d'un programme permet de comparer commodément le fonctionnement de plusieurs programmes entre eux ou d'un même programme avec des paramètres différents. La comparaison entre programmes est facilitée par l'affichage de plusieurs sessions dans la fenêtre *Sessions*. Il suffit ensuite de cliquer sur les *Context_0*, *Device_0* et *Session1* ou *Session2* pour comparer très rapidement les caractéristiques de deux programmes.

Lors de l'exécution du profilage, on remarque qu'un même programme peut être exécuté plusieurs fois. Le nombre d'exécutions dépend du nombre de paramètres choisis dans les options de la session (*Session Settings*). Etant donné que seule une partie des compteurs de profil choisis (paramètres) peuvent être activés lors d'une seule exécution de programme, le programme doit être exécuté plusieurs fois. Ci-dessus, on peut observer un profilage type.

## 3.3. Premiers débogueurs NVIDIA: Nexus et Parallel Nsight

Depuis 2009, il existe des logiciels de débogage complètement intégrés à Visual Studio. Ce sont *Nexus* et *Parallel Nsight* : sorti en 2009, Nexus sera rapidement remplacé par *Parallel Nsight* en 2010.

Sorti pour la première fois en 2009, *NVIDIA Nexus* est le premier environnement de développement spécialement conçu pour prendre en charge le CUDA-C dans des applications parallèles. Il fait le lien entre code CPU et code GPU en intégrant le débogage de code source et l'analyse de données directement dans Visual Studio.

*Nexus* permet aux développeurs Visual Studio d'écrire et de déboguer du code source GPU en utilisant exactement les mêmes outils et interfaces que pour l'écriture et le débogage de code source CPU, y compris les points d'arrêt, et l'inspection de la mémoire. De plus, *Nexus* étend les fonctionnalités de Visual Studio en offrant des outils de gestion du parallélisme, tels que la possibilité de se concentrer sur une tâche unique (parmi les milliers de tâches fonctionnant en parallèle) et la déboguer, et la possibilité de visualiser simplement et efficacement les résultats produits par l'ensemble des tâches parallèles.

Attention : Nexus n'est compatible qu'avec Windows Vista SP1 ou avec Windows 7.

Dès janvier 2010, un nouvel outil de débogage sort pour le développement d'applications parallèles : c'est l'outil de débogage et de profilage *NVIDIA Parallel Nsight* disponible comme un module complémentaire pour Visual Studio.

Comme pour *Nexus*, *Parallel Nsight* n'est compatible qu'avec les systèmes Windows Vista et Windows 7.

A savoir qu'il existe un autre débogueur CUDA mais utilisable sur Linux uniquement, c'est CUDA-GDB.

# 4. Améliorations possibles

Dans ce mémoire une grande partie des possibilités de CUDA™ a été exploitées. Cependant, par manque de temps, quelques fonctionnalités n'ont pas pu l'être. Le but de cette partie est donc de présenter rapidement les différentes possibilités d'améliorations.

## 4.1. Notion de flux de données ou streaming

L'utilisation de flux est très utile pour pouvoir exécuter des tâches complètement différentes en parallèle. L'utilisation de flux peut permettre d'exécuter deux kernels simultanément avec par exemple :

- un des deux kernels qui effectue *n* additions simultanées dans une grille de tâches et
- l'autre kernel qui effectue *m* multiplications simultanées dans une autre grille de tâches.

Ceci n'aurait pas été possible sans l'existence de la notion de flux.

L'utilisation de flux est également possible sous CUBLAS, CUFFT, CURAND et CUSPARSE.

## 4.2. Préchargement de données (prefetching)

Une des plus importantes limitations de ressources pour l'informatique parallèle en général est le fait que la mémoire globale a une bande passante limitée lors de l'accès aux données, et ces accès demandent beaucoup de temps à s'achever.

Une solution utile au problème est de faire un préchargement des données opérandes suivantes tout en utilisant les données courantes déjà chargées, ce qui accroit le nombre d'instructions indépendantes entre les accès mémoire et l'utilisation des données accédées.

La technique de préchargement de données est souvent utilisée dans les boucles d'itérations. L'avantage de précharger des données est de tirer profit de l'aspect asynchrone de l'accès mémoire dans CUDA™. Lorsqu'une opération d'accès à la mémoire est exécutée, elle ne bloque pas les autres opérations qui suivent à condition que ces dernières n'utilisent pas les données en train d'être chargées.

## 4.3. Déroulement de boucles

Dans certains processeurs graphiques CUDA, chaque cœur a une bande passante de traitement d'instructions limitée. Chaque instruction utilise de la bande passante de traitement d'instruction, que ce soit pour :

- une instruction de calcul en virgule flottante,
- une instruction de chargement,
- ou une instruction de branchement.

Dans le cas de la multiplication de matrices, la boucle du produit scalaire utilise des instructions supplémentaires pour remettre à jour le compteur de boucle et effectuer des branchements conditionnels à la fin de chaque itération.

Une méthode souvent utilisée pour améliorer le ratio entre instructions de calcul et autres instructions est de dérouler la boucle.

Le déroulement permet d'éliminer l'instruction de branchement et la remise à jour de compteur de boucle. Idéalement, le déroulement de boucle est réalisé automatiquement par le compilateur. Comme pour la technique du préchargement de données, le déroulement de boucle peut nécessiter des registres supplémentaires pour stocker les données équivalentes à de multiple passage dans la boucle et par conséquent limiter le nombre de blocs exécutables par multiprocesseur.

## 4.4. Améliorations diverses

- Lorsqu'on veut réaliser la division par deux d'un nombre, il est conseillé de faire à la place un décalage à droite du nombre sous sa forme binaire.
- Lorsqu'on veut réaliser la multiplication par deux d'un nombre, il est conseillé de faire à la place un décalage à gauche du nombre sous sa forme binaire.
- En mode débogage, il peut parfois être utile d'appeler les fonctions CUDA par l'intermédiaire de la macro `CUDA_SAFE_CALL`. Cette macro vérifie si la fonction appelée ne renvoie d'erreurs. Attention, il ne faut pas oublier de définir *_DEBUG* dans la ligne de commande (voir l'annexe sur la configuration d'un fichier et d'un projet CUDA) car sinon cette macro n'est pas fonctionnelle pour des raisons de performance. Pour pouvoir utiliser `CUDA_SAFE_CALL`, il ne faut pas non plus oublier d'ajouter la fonction d'entête *cutil.h* (fournie avec CUDA™) en début du programme CUDA.

# Annexe 6
# Programme MEX-CUDA-C

```
/* Programme utilisant la mémoire partagée */
/* Utilisation de nombres flottants double précision */

#include <stdio.h>
#include "mex.h"
#include <cuda.h>

typedef struct {
int width;
int height;
int stride;
double* elements;
} Matrix;

/* Nombre de tâches par ligne (dans un bloc) */
#define BLOCK_SIDE 16

/* Extraction d'un élement de matrice */
__device__ double GetElement(const Matrix A, int row, int col)
        {
        return A.elements[row * A.stride + col];
        }

/* Initialisation d'un élément de matrice */
__device__ void SetElement(Matrix A, int row, int col, double value)
        {
        A.elements[row * A.stride + col] = value;
        }

        /* Extrait la sous-matrice Asub de dimensions BLOCK_SIDE x
BLOCK_SIDE (à partir de la matrice A) et située col sous-matrices en
partant de la gauche et row sous-matrices vers le bas en partant du
coin supérieur gauche de A */
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
        {
        Matrix Asub;
        Asub.width = BLOCK_SIDE;
        Asub.height = BLOCK_SIDE;
        Asub.stride = A.stride;
        /* Extrait l'adresse du premier élément de chaque bloc (dans
l'ordre des blocs) */
        Asub.elements = &A.elements[A.stride * BLOCK_SIDE * row +
BLOCK_SIDE * col];
        return Asub;
        }


/* Déclaration du kernel de multiplication matricielle */
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

        /* Code hôte de la multiplication matricielle. Les dimensions de
matrice sont supposées être des multiples de BLOCK_SIDE */
void mexFunction( int nlhs, mxArray *plhs[],int nrhs, const mxArray
*prhs[])
        {
```

```
        /* Déclaration de pointeurs vers les matrices d'entrée */
        double *h_A,*h_B,*h_C;

        unsigned int mrowsA, ncolsA, mrowsB, ncolsB; /* entier non signé
*/

        /* Vérifie que le nombre d'arguments d'entrée et de sortie sont
corrects */
        if(nrhs!=2) mexErrMsgTxt("Deux entrées demandées.");
        if(nlhs!=1) mexErrMsgTxt("Une sortie seulement demandée.");

        /* Extraction des dimensions des matrices d'entrée */
        mrowsA = (unsigned int)mxGetM(prhs[0]);
        ncolsA = (unsigned int)mxGetN(prhs[0]);
        mrowsB = (unsigned int)mxGetM(prhs[1]);
        ncolsB = (unsigned int)mxGetN(prhs[1]);

        /* Crée un pointeur pour la matrice hôte d'entrée */
        h_A = mxGetPr(prhs[0]);
        h_B = mxGetPr(prhs[1]);

        /* Chargement de A et B en mémoire GPU */
        Matrix d_A;
        d_A.width = d_A.stride = ncolsA; d_A.height = mrowsA;
        size_t size = ncolsA * mrowsA * sizeof(double);

        cudaMalloc((void**)&d_A.elements, size);
        cudaMemcpy(d_A.elements, h_A, size, cudaMemcpyHostToDevice);

        Matrix d_B;
        d_B.width = d_B.stride = ncolsB; d_B.height = mrowsB;
        size = ncolsB * mrowsB * sizeof(double);

        cudaMalloc((void**)&d_B.elements, size);
        cudaMemcpy(d_B.elements, h_B, size, cudaMemcpyHostToDevice);

        /* Allocation de mémoire pour C en mémoire GPU */
        Matrix d_C;
        d_C.width = d_C.stride = ncolsB; d_C.height = mrowsA;
        size = ncolsB * mrowsA * sizeof(double);

        cudaMalloc((void**)&d_C.elements, size);

        dim3 dimBlock(BLOCK_SIDE, BLOCK_SIDE);
        dim3 dimGrid(ncolsB / dimBlock.x, mrowsA / dimBlock.y);

        /* Mesure du temps d'exécution */
        /*cudaEvent_t start, stop;
        float time;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord( start, 0 );*/ /* Début */

        /* Appel du kernel */
        MatMulKernel<<<dimGrid, dimBlock>>>(d_B, d_A, d_C);

        /* Fin de la mesure du temps d'exécution du programme */
        /*cudaEventRecord( stop, 0 );*/ /* Fin */
        /*cudaEventSynchronize( stop );
        cudaEventElapsedTime( &time, start, stop );
```

```
        cudaEventDestroy( start );
        cudaEventDestroy( stop );*/

        /* Affichage du temps d'exécution sur carte graphique */
        /*printf("\n");
        printf("Temps ecoule: %f ms\n", time);*/

        /* Crée un mxArray 2 dimensions pour des flottants double
précision initialisés à 0 */
        /* Initialise le pointeur de sortie sur la matrice de sortie */
        plhs[0] = mxCreateDoubleMatrix(mrowsA,ncolsB, mxREAL);

        /* Crée un pointeur pour la matrice hôte de sortie */
        h_C = mxGetPr(plhs[0]);

        /* Transfert du résultat de la carte graphique vers le CPU */
        cudaMemcpy(h_C, d_C.elements, size, cudaMemcpyDeviceToHost);

        /* Libération de la mémoire GPU */
        cudaFree(d_A.elements);
        cudaFree(d_B.elements);
        cudaFree(d_C.elements);

        cudaThreadExit();
        }

/* Kernel de multiplication matricielle appelé par le programme
principal (mexFunction) */
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
        {
        /* Extraction dans un bloc des numéros de ligne et des numéros
de colonnes */
        int blockRow = blockIdx.y;
        int blockCol = blockIdx.x;

        /* Chaque bloc de tâches calcule une sous-matrice Csub de C */
        Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

        /* Chaque tâche calcule un élément de Csub en cumulant les
résultats dans Cvalue */
        double Cvalue = 0;

        /* Extraction des indices rangée-colonne des tâches à
l'intérieur de Csub */
        int row = threadIdx.y;
        int col = threadIdx.x;

        /* Boucles sur chaque paire de blocs (de A et B) requise pour
calculer Csub. Produit de chaque paire de blocs et accumulation des
résultats */
        for (int m = 0; m < (A.width / BLOCK_SIDE); ++m) {
                /* Extraction du bloc Asub à partir de A */
                Matrix Asub = GetSubMatrix(A, blockRow, m);

                /* Extraction du bloc Bsub à partir de B */
                Matrix Bsub = GetSubMatrix(B, m, blockCol);

                /* Réservation de mémoire partagée pour stocker Asub et
Bsub respectivement */
                __shared__ double As[BLOCK_SIDE][BLOCK_SIDE];
                __shared__ double Bs[BLOCK_SIDE][BLOCK_SIDE];
```

```
                /* Chargement de Asub et Bsub de la mémoire GPU vers la
mémoire partagée. Chaque tâche charge un élément de chaque bloc. */

                As[row][col] = GetElement(Asub, row, col);
                Bs[row][col] = GetElement(Bsub, row, col);

                /* Synchronisation pour s'assurer que les deux blocs sont
complètement chargées avant le démarrage du calcul */
                __syncthreads();

                /* Produit scalaire d'une ligne d'éléments de Asub et
d'une colonne d'éléments de Bsub */
                for (int e = 0; e < BLOCK_SIDE; ++e)
                        Cvalue += As[row][e] * Bs[e][col];

                /* Synchronisation pour s'assurer que le calcul précédent
est terminé avant de charger deux nouveaux blocs de A et B lors les
itérations suivantes */
                __syncthreads();
                }

        /* Ecriture de Csub dans la mémoire globale de la carte. Chaque
tâche écrit un élément résultat */
        SetElement(Csub, row, col, Cvalue);
        }
```

Le programme MATLAB de vérification des résultats et des erreurs de calcul est le suivant :

```
t=1024;  %Taille des côtés (nombre d'éléments) des matrices
carrées

x1=rand(t,t);

x2=rand(t,t);

y=x1*x2;

z=mulmat17(x1,x2);


%Vérification de l'importance et du nombre des erreurs sur un nombre
réduit de éléments de la matrice résultat
rescudamex=z(1:8,1:8)
resmat=y(1:8,1:8)
diff=resmat-rescudamex

%Vérification de l'importance et du nombre des erreurs sur la totalité
des éléments de la matrice résultat
diff2=y-z;
hist(diff2(:),200)

clear global;
```

Le programme MATLAB de mesure des temps est le suivant :

```
t=1024;
x1=rand(t,t);
x2=rand(t,t);

% Mesure des temps de calcul de MATLAB seul sur 10 valeurs
tic;
```

```
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
toc

% L'exécution de la 1ère des instructions suivantes n'est pas
comtabilisée en raison des incertitudes dues au temps de mise en place
des ressources sur la carte graphique
z=mulmat17(x1,x2);
% Mesure des temps de calcul de la fonction MEX-CUDA-C sur 10 valeurs
tic;
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
z=mulmat17(x1,x2);
toc

clear global;
```

Le programme MATLAB de test appelé par le profiler est le suivant :

```
t=1024;
x1=rand(t,t);
x2=rand(t,t);

z=mulmat17(x1,x2);

clear global;
quit;
```

# Annexe 7
# Programme MEX-CUBLAS-C

```
/* Programme de multiplication de matrices avec nombres à virgule
flottante double précision */

/* mulmat05.cpp : définit le point d'entrée pour l'application console.
*/

#include <stdio.h>
#include <stdlib.h>
#include "mex.h"
#include "cuda.h"
#include "cublas.h"

/* Calcul de l'indice de rangement en colonne (dans un vecteur à indice
débutant à 0) de l'élément de la ième ligne et de la jème colonne d'un
tableau à indices débutant à 0 */
#define IDX2C(i,j,ld) (((j)*(ld))+(i))

void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray
*prhs[])
{

cublasStatus status = 0; /* cublasStatus est un type prédéfini comme
entier non signé dans la bibliothèque cublas.h  */
status = cublasInit(); /* Initialisation de la bibliothèque CUBLAS */

if (status != CUBLAS_STATUS_SUCCESS)
        {
        printf("status = %d", status);
        mexErrMsgTxt("Erreur lors de l'initialisation de CUBLAS !\n");
        }

/* Déclaration des variables */
double *h_A,*h_B,*h_C;
double *d_A,*d_B,*d_C;
double alpha = 1;
double beta = 0;
int i;
unsigned int mrowsA, ncolsA, mrowsB, ncolsB; /* entier non signé */

/* Vérifie que le nombre d'arguments d'entrée et de sortie sont
corrects */
if(nrhs!=2) mexErrMsgTxt("Deux entrées demandées.");
if(nlhs!=1) mexErrMsgTxt("Une sortie seulement demandée.");

/* Extraction des dimensions des matrices d'entrée */
mrowsA = (unsigned int)mxGetM(prhs[0]);
ncolsA = (unsigned int)mxGetN(prhs[0]);
mrowsB = (unsigned int)mxGetM(prhs[1]);
ncolsB = (unsigned int)mxGetN(prhs[1]);

/* Crée un pointeur pour la matrice hôte d'entrée */
h_A = mxGetPr(prhs[0]);
h_B = mxGetPr(prhs[1]);

int nb_elem1 = mrowsA * ncolsA;
```

```
int nb_elem2 = mrowsB * ncolsB;
int nb_elem3 = mrowsA * ncolsB;

/* Crée un mxArray 2 dimensions pour des flottants double précision
initialisés à 0. Initialise le pointeur de sortie sur la matrice de
sortie */
plhs[0] = mxCreateDoubleMatrix(mrowsA,ncolsB, mxREAL);

/* Crée un pointeur pour la matrice hôte de sortie */
h_C = mxGetPr(plhs[0]);

/* Remplissage de la matrice de sortie par des zéros */
for (i = 0 ; i < nb_elem3 ; i++)
       {
       h_C[i] = 0;
       }

/* Allocation de la mémoire GPU avec gestion des erreurs */
/* d_A */
status = cublasAlloc(nb_elem1, sizeof(d_A[0]), (void **) & d_A);
if (status != CUBLAS_STATUS_SUCCESS)
{
       printf("status = %d", status);
       mexErrMsgTxt("Erreur lors de l'allocation mémoire de la matrice
A sur le périphérique !\n");
}

/* d_B */
status = cublasAlloc(nb_elem2, sizeof (d_B[0]), (void **) & d_B);
if (status != CUBLAS_STATUS_SUCCESS)
       mexErrMsgTxt("Erreur lors de l'allocation mémoire de la matrice
B sur le périphérique !\n");

/* d_C */
status = cublasAlloc(nb_elem3, sizeof (d_C[0]), (void **) & d_C);
if (status != CUBLAS_STATUS_SUCCESS)
       mexErrMsgTxt("Erreur lors de l'allocation mémoire de la matrice
C sur le périphérique !\n");

/* Remplissage des matrices du périphérique avec les données des
matrices de l'hôte et gestion des erreurs */
status = cublasSetVector(nb_elem1, sizeof(h_A[0]), h_A, 1, d_A, 1);
if (status != CUBLAS_STATUS_SUCCESS)
       mexErrMsgTxt("Erreur d'accès au périphérique lors de l'écriture
de la matrice A !\n");

status = cublasSetVector(nb_elem2, sizeof(h_B[0]), h_B, 1, d_B, 1);
if (status != CUBLAS_STATUS_SUCCESS)
       mexErrMsgTxt("Erreur d'accès au périphérique lors de l'écriture
de la matrice B !\n");

/* La fonction suivante renvoie la dernière erreur apparue à l'appel de
n'importe laquelle des fonctions coeur de CUBLAS. Contrairement aux
fonctions "helper" de CUBLAS, les fonctions coeur de CUBLAS ne
renvoient pas le statut directement. */
status = cublasGetError();

/* Mesure du temps d'exécution */
       /*cudaEvent_t start, stop;
       float time;
       cudaEventCreate(&start);
```

```
        cudaEventCreate(&stop);
        cudaEventRecord( start, 0 );*/ /* Début */

/* Multiplication des matrices avec CUBLAS en nombres flottants double
précision*/
cublasDgemm('n', 'n', mrowsA, ncolsB, ncolsA, alpha, d_A, mrowsA, d_B,
mrowsB, beta, d_C, mrowsA);

/* Fin de la mesure du temps d'exécution du programme */
        /*cudaEventRecord( stop, 0 );*/ /* Fin */
        /*cudaEventSynchronize( stop );
        cudaEventElapsedTime( &time, start, stop );
        cudaEventDestroy( start );
        cudaEventDestroy( stop );*/

        /* Affichage du temps d'exécution sur la carte graphique */
        /*printf("\n");
        printf("Temps ecoule: %f ms\n", time);*/

/* Lecture du résultat avec gestion des erreurs */
status = cublasGetVector(nb_elem3, sizeof(h_C[0]), d_C, 1, h_C, 1);
if (status != CUBLAS_STATUS_SUCCESS)
        mexErrMsgTxt("Erreur d'accès au périphérique lors de la lecture
de la matrice C !\n");

/* Libération de la mémoire sur le périphérique avec gestion des
erreurs */
status = cublasFree(d_A);
if (status != CUBLAS_STATUS_SUCCESS)
        mexErrMsgTxt("Erreur lors de la libération de l'espace
périphérique de la matrice A !\n");

status = cublasFree(d_B);
if (status != CUBLAS_STATUS_SUCCESS)
        mexErrMsgTxt("Erreur lors de la libération de l'espace
périphérique de la matrice B !\n");

status = cublasFree(d_C);
if (status != CUBLAS_STATUS_SUCCESS)
        mexErrMsgTxt("Erreur lors de la libération de l'espace
périphérique de la matrice C !\n");

/* Fermeture de CUBLAS avec gestion des erreurs. Libère les resources
CPU utilisées par la bibliothèque CUBLAS. */
status = cublasShutdown();
if (status != CUBLAS_STATUS_SUCCESS)
        mexErrMsgTxt("Erreur à la fermeture de CUBLAS !\n");

/*cudaThreadExit();*/
cublasShutdown();
}
```

Le programme MATLAB de vérification des résultats et des erreurs de calcul est le suivant :

```
t=1024; %Taille des côtés (nombre d'éléments) des matrices carrées
x1=rand(t,t);
x2=rand(t,t);

y=x1*x2;
```

```
z=mulmat05(x1,x2);

%Vérification du nombre d'erreurs et de leur importance sur un nombre
réduit de éléments de la matrice résultat
rescudamex=z(1:8,1:8)
resmat=y(1:8,1:8)
diff=resmat-rescudamex

%Vérification du nombre d'erreurs et de leur importance sur la totalité
des éléments de la matrice résultat
diff2=y-z;
hist(diff2(:),200)

clear global;
```

Le programme MATLAB de mesure des temps est le suivant :

```
t=1024;
x1=rand(t,t);
x2=rand(t,t);

% Mesure des temps de calcul de MATLAB seul sur 10 valeurs
tic;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
y=x1*x2;
toc

% L'exécution de la 1ère des instructions suivantes n'est pas
comptabilisée en raison des incertitudes dues au temps de mise en place
des ressources sur la carte graphique
z=mulmat05(x1,x2);
% Mesure des temps de calcul de la fonction MEX-CUDA-C sur 10 valeurs
tic;
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
z=mulmat05(x1,x2);
toc

clear global;
```

Le programme MATLAB de test appelé par le profiler est le suivant :

```
t=1024;
x1=rand(t,t);
x2=rand(t,t);
```

```
z=mulmat05(x1,x2);

clear global;
quit;
```

# Annexe 8
# Programme MEX-CUFFT-C

```
/* Programme MEX-CUFFT pour un vecteur */

#include <stdio.h>
#include <math.h>
#include "mex.h"
#include <cuda.h>
#include <cuda_runtime.h>
#include <cufft.h>


void pack_dc2dc(cufftDoubleComplex  *output, double *input_re, double
*input_im, int Ntot)
        {
        int n;
        for(n=0; n<Ntot; n++)
                {
                output[n].x = input_re[n];
                output[n].y =  input_im[n];
                }
        }

void unpack_dc2dc(cufftDoubleComplex *input, double *output_re, double
*output_im, int Ntot)
        {
        int n;
        for(n=0; n<Ntot; n++)
                {
                output_re[n] = input[n].x;
                output_im[n] = input[n].y;
                }
        }

void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray
*prhs[])
        {
        /* Déclaration des variables d'entrée et des pointeurs */
        cufftDoubleComplex *h_A, *h_B;
        double *prA, *piA;
        unsigned int nx, batch; /* entier non signé */

        /* Vérification du nombre d'arguments d'entrée et de sortie */
        if(nrhs!=1) mexErrMsgTxt("Une entrée demandée.");
        if(nlhs!=1) mexErrMsgTxt("Une sortie seulement demandée.");

        /* Extraction des dimensions des matrices d'entrée */
        nx = (unsigned int)mxGetM(prhs[0]);
        batch = (unsigned int)mxGetN(prhs[0]);

        prA = mxGetPr(prhs[0]);
        piA = mxGetPi(prhs[0]);

        /* Réservation d'espace mémoire */
        h_A = (cufftDoubleComplex*)mxMalloc(sizeof(cufftDoubleComplex) *
nx * batch);
```

```
        h_B = (cufftDoubleComplex*)mxMalloc(sizeof(cufftDoubleComplex) *
nx * batch);

        cufftHandle plan;
        cufftDoubleComplex *devPtr;

        /* Transformation du format Re/Im de Matlab en un tableau
cufftDoubleComplex */
        pack_dc2dc(h_A,prA,piA,nx*batch);

        /* Allocation de mémoire GPU */
        cudaMalloc((void**)&devPtr,
sizeof(cufftDoubleComplex)*nx*batch);

        /* Transfert en mémoire GPU */
        cudaMemcpy(devPtr, h_A, sizeof(cufftDoubleComplex)*nx*batch,
cudaMemcpyHostToDevice);

        /* Création d'un plan FFT 1D */
        cufftPlan1d(&plan, nx, CUFFT_Z2Z, batch); /* nx: taille de la
transformée */

        /* Exécution d'une FFT directe */
        cufftExecZ2Z(plan, devPtr, devPtr, CUFFT_FORWARD);

        /* Exécution d'une FFT inverse */
        /*cufftExecC2C(plan, devPtr, devPtr, CUFFT_INVERSE);*/

        /* Transfert des résultats à partir de la mémoire GPU */
        cudaMemcpy(h_B, devPtr, sizeof(cufftDoubleComplex)*batch*nx,
cudaMemcpyDeviceToHost);

        /* Initialisation du pointeur de sortie sur la matrice de sortie
*/
        plhs[0] = mxCreateDoubleMatrix(nx,batch, mxCOMPLEX);

        /* Transformation d'un tableau cufftComplex en un format Re/Im
compatible Matlab */
        unpack_dc2dc(h_B,mxGetPr(plhs[0]),mxGetPi(plhs[0]),nx*batch);

        /* Suppression du plan CUFFT */
        cufftDestroy(plan);

        /* Libération de la mémoire GPU */
        cudaFree(devPtr);

        /* Libération de la mémoire CPU */
        mxFree(h_A);
        mxFree(h_B);
        }
```

Le programme MATLAB de vérification des résultats et des erreurs de calcul est le suivant :

```
nx=1024; %Taille de la transformée de Fourier
batch=1024; %Nombre de transformées de meme taille

x=rand(nx,batch) + i*rand(nx,batch);

%FFT MATLAB
y=fft(x);
```

```
%FFT CUDA
z=cfft02(x);

%Vérification du nombre d'erreurs et de leur importance sur un nombre
réduit de éléments de la matrice résultat
resmat=y(1:4,1:4)
rescudamex=z(1:4,1:4)
diff=resmat-rescudamex

%Vérification du nombre d'erreurs et de leur importance sur la totalité
des éléments de la matrice résultat
diff2=y-z;
prd2 = real(diff2); %Extraction de la partie réelle de diff2
pid2 = imag(diff2); %Extraction de la partie imaginaire de diff2

hist(prd2(:),256)
hist(pid2(:),256)
clear global;
```

Le programme MATLAB de mesure des temps est le suivant :

```
nx=1024; %taille de la transformée de Fourier
batch=1024; %nombre de transformées de meme taille
x=rand(nx,batch) + i*rand(nx,batch);

% Mesure des temps de calcul de MATLAB seul sur 10 valeurs
tic;
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
y=fft(x);
toc

% L'exécution de la 1ère des instructions suivantes n'est pas
comtabilisée en raison des incertitudes dues au temps de mise en place
des ressources de la carte graphique
z=cfft02(x);
% Mesure des temps de calcul de la fonction MEX-CUDA-C sur 10 valeurs
tic;
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
z=cfft02(x);
toc
clear global;
```

Le programme MATLAB appelé par le profiler est le suivant :

```
nx=1024; %taille de la transformée de Fourier
batch=1024; %nombre de transformées de meme taille
x=rand(nx,batch) + i*rand(nx,batch);
z=cfft02(x);
clear global;
quit;
```

# Liste des figures

# Liste des tableaux

## Résumé

Ce mémoire présente l'étude des capacités de calcul d'une carte graphique dernière génération pour observer dans quelle mesure elle peut se substituer au CPU pour les calculs parallèles. Ce mémoire présente également une possibilité d'application du calcul GPU pour du traitement de signal radar.

Pour atteindre l'objectif ci-dessus, nous avons procédé à l'étude approfondie de trois API dédiées aux calculs sur carte graphique NVIDIA : CUDA, CUBLAS et CUFFT.

Pour réaliser cette étude, nous avons utilisé une carte graphique GTX470 du fabricant NVIDIA comportant 448 cœurs de traitement parallèle.

Le logiciel MATLAB a été utilisé afin de vérifier l'exactitude des calculs GPU. Une fonction MEX a été utilisée pour que MATLAB et le code CUDA puissent se transmettre des données.

Cette étude a été réalisée avec les outils logiciels MATLAB 7.1, Visual C++ (Visual Studio 2005) et les outils spécifiques CUDA que sont le Profiler et tableau de calcul de taux d'occupation des ressources.

## Abstract

This document presents the study of the compute resources of a video card from the most recent ones in order to observe to which degree it is able to replace the CPU for parallel computing. This document also presents an example of implementation of GPU computing for radar signal processing.

In order to achieve this goal, we carried out the complete study of three APIs dedicated to computing on NVIDIA video cards : CUDA, CUBLAS and CUFFT.

For this study, we used a GTX470 video card made by NVIDIA and featuring 448 streaming processors for parallel computing.

The MATLAB software has been used in order to check the accuracy of the GPU computing. A MEX function has been used in order for MATLAB and the CUDA program to be able to exchange data.

This study has been carried out using the MATLAB 7.1 and Visual C++ (Visual Studio 2005) softwares as well as the CUDA dedicated tools which are the Compute Visual Profiler and the Occupancy Calculator.

**Mots-clés :** GPU, GTX 470, SIMT, CUDA, CUBLAS, CUFFT, C, processeur graphique, multiprocesseur, cœur de traitement, mémoire partagée, calcul parallèle.

**Keywords** : GPU, GTX 470, SIMT, CUDA, CUBLAS, CUFFT, C, graphic processor, streaming multiprocessor, streaming processor, shared memory, parallel computing.