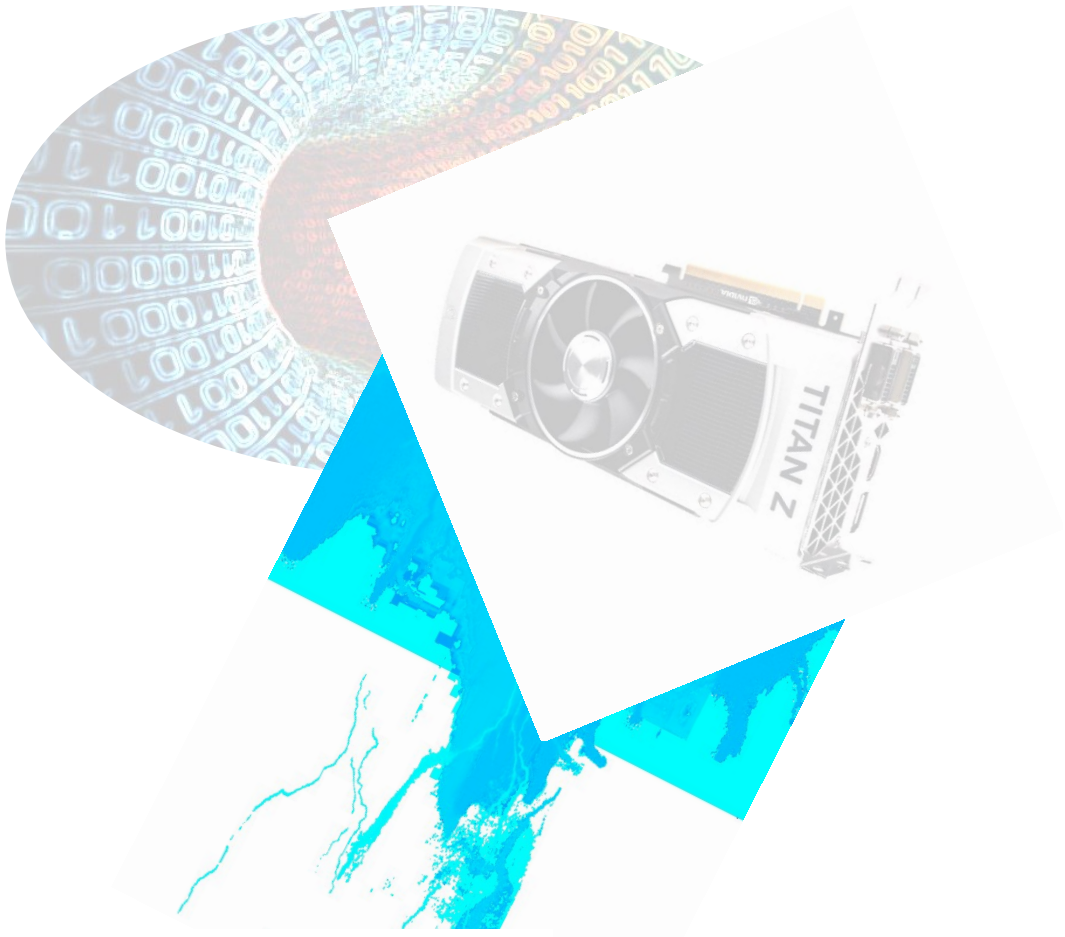
ÉTS
Le génie pour l'industrie

Introduction au « GPU computing »

Architecture et programmation d'une carte graphique

GUYON Mathias, TETREL Loïc



Projet de cours de SYS 809 : Vision par ordinateur

Enseigné par Mr Richard Lepage

Table des matières

Résumé

Ce rapport présente un compte rendu de notre projet du cours SYS809 : Vision par ordinateur qui a duré 10 semaines et qui concerne le GPU computing. Ce domaine consiste à utiliser les ressources des cartes graphiques pour accélérer le temps d'exécution de diverses applications. En effet, nous arrivons dans une ère de l'informatique où la puissance des CPU n'augmente plus, les ingénieurs se tournent alors vers les GPU qui en exécutant des tâches en parallèle parviennent à améliorer les performances de leurs applications. L'objectif du projet est la découverte du GPU computing, la maîtrise de ces concepts de base et l'analyse par nous-même de ces caractéristiques si avantageuses.

Nous avons donc étudié la technologie des cartes graphiques du point de vue matériel et approché les concepts de programmation hétérogènes (CPU+GPU) et parallèle. Une initiation à la plateforme de programmation CUDA développée par le constructeur de GPU NVIDIA nous a permis de programmer nos propres applications. Le cœur du projet est l'analyse de performance de ces applications et la comparaison avec les performances d'un CPU seul. Nous avons utilisé les cartes graphiques NVIDIA présentent dans nos ordinateurs personnels. L'analyse de performance a pu mettre en valeur la capacité des GPU à effectuer de grandes quantités de calculs en parallèle. Nous avons choisis comme cas d'étude la multiplication matricielle, exemple classique permettant de mettre en œuvre le parallélisme, et l'implémentation d'un filtre Sobel sur une image. L'accélération de ces applications a atteint dans les deux cas plus de 99%.

Introduction

La loi de Moore[1] illustre l'évolution du nombre de transistors dans les microprocesseurs qui est sensé doubler tous les 18 mois. On considère qu'un corolaire de cette loi est que la vitesse des microprocesseurs évolue de la même façon. Cependant, nous arrivons à un moment de l'histoire de l'informatique où cette évolution n'est plus possible et ce pour plusieurs raisons. Dans un premier temps, plus de transistors signifie des transistors plus petits qui dissipent toujours plus de chaleur. Il en résulte également des microprocesseurs qui consomment bien trop de puissance. Nous pouvons noter enfin que la vitesse maximale des mémoires est déjà largement dépassée par celle des CPU. Ces contraintes demandent des efforts d'une telle complexité que la vitesse gagnée n'en vaut plus la peine. Dans ce contexte, ce ne sont plus les plus rapide qui comptent mais les plus larges, ceux qui compte le plus de cœurs, dont les capacités à effectuer des opérations en parallèle prennent toute leur importance. Dans le même temps, l'industrie du Jeu Vidéo a développée des cartes graphiques capables de générer des images de très grande résolution avec des couleurs représentant au mieux le monde réel et ce, 60 fois par seconde ! Les GPU y sont utilisées pour générer des mouvements complexes reproduits en vidéo comme la dissipation de fumée, des explosions, l'expression des visages, … Nous sommes alors en présence d'un besoin de parallélisme face aux limites des CPU contemporains et de cartes graphiques ultraperformantes. C'est l'émergence du « General Purpose GPU Computing » qui consiste à utiliser les ressources des cartes graphiques pour des opérations non destinées au graphisme.

Ce projet consiste à nous initier à la programmation des cartes graphiques, très populaire dans l'industrie depuis quelques années. En tant qu'introduction à ce domaine, notre sujet nous fera découvrir les bases du parallélisme, manipuler et expérimenter ses outils et observer les performances des cartes graphiques pour des applications dédiées. Pour ne pas trop s'éloigner du contexte du cours, l'objectif que nous nous sommes fixé est de filtrer une image mais en utilisant les cartes graphiques que nous possédons dans nos ordinateurs personnels. La question que nous nous posons est : comment, à quel point et à quel prix le GPU Computing peut-il accélérer une application ? Ce rapport présente un compte-rendu de notre étude qui a duré 10 semaines. Nous vous offrons tout d'abord un état de l'art du sujet étudié comme mise en contexte. Ensuite, nous introduirons les notions et concepts que nous jugeons utiles pour appréhender les cartes graphiques, leur matériel et les différentes façons de les programmer. Enfin nous présenterons le cœur de notre travail, une analyse des performances GPU contre CPU lors de deux cas d'étude.

[1] Par Gordon Moore dans *Electronics Magazine* en 1965.

Revue de littérature

Avant de nous attarder sur le projet, il est nécessaire de se pencher sur l'art de ce que l'on appelle le « GPU computing ». Nous allons voir que malgré la modernité de ce domaine, l'idée d'utiliser les unités graphiques en tant que processeur de calcul vient d'il y a déjà quelques années. Nous nous pencherons sur la différente évolution majeures et les diverses manières de répondre à la question de l'utilisation du GPU.

Un des premiers articles scientifiques traitant du sujet est proposé par Barron et Glorioso en 1973 (1), ils utilisent ainsi une carte graphique qui travaille en association avec un processeur pour gérer l'affichage d'un écran. C'est une des premières pierres du « GPU computing ». Peu à peu, l'idée d'utiliser le GPU comme processeur à germer dans la tête des nombreux chercheurs, la problématique du temps de traitement étant en effet toujours importante. Une des applications majeures arriva en 1985, Altaber et al. travaillent alors sur le stockage des Large Electron Positron (LEP) dans le système de contrôle de l'anneau du CERN. Ils utilisent des GPU (2) pour accélérer le traitement des électrons dans l'anneau en utilisant une multitude de cartes graphiques travaillant en parallèle. Ils concluent que remplacer les CPU par des GPU améliore de façon exponentielle la puissance de calcul. Un peu plus tard dans les années 2000, Shen et al. (3) utilisent des GPU pour accélérer le décodage vidéo qui était auparavant fait pas des CPU. Ils discutent plus précisément de la consommation énergétique des cartes graphiques pour conclure que : Contrairement à des idées préconçues, la consommation des GPU n'est pas un critère discriminant pour leurs utilisations. Dans la même année, Krüger et Westermann (4) implémente une bibliothèque d'algèbre linéaire en C/C++ et démontrent les capacités du GPU qui est alors utilisé pour des traitements numériques. C'est le début de l'ouverture de la programmation sur carte graphique au « grand public ». C'est ce qui donnera des idées à l'un des plus grands constructeurs de carte graphique au monde, NVIDIA. En effet, par le passé le GPU computing n'était accessible qu'en modifiant le matériel et était compliqué à implémenter.

En 2006, NVIDIA propose au grand public la première carte graphique au monde permettant d'effectuer du GPGPU (General-purpose Processing on Graphics Processing Units) en langage standard. Il s'agit de la GeForce 8800 profitant de l'architecture TESLA qui offre une architecture unifiée des différents cœurs, cœurs appelés « CUDA[2] Core ». Ces cœurs sont utilisés dynamiquement par l'intelligence de la carte et permet la programmation par carte graphique à l'aide du compilateur « NVIDIA C-compiler ». Ce nouveau paradigme fut officialisé par le biais d'une conférence sur l'optimisation de code en

[2] Il est souvent dit que CUDA est l'abréviation de « Compute Unified Device Architecture » ce qui est complètement faux d'après la conférence en 2012 de Mark Ebersole. Il s'agirait d'un hommage à l'un des ingénieurs ayant travaillé sur le projet et qui avait un coupé « Plymouth Barracuda » surnommé… Cuda !

2007 par Buck (5). Suite à cette conférence d'une envergure internationale, on assiste à l'explosion du GPU computing que ce soit dans l'industrie ou dans la recherche. Un peu plus tard en 2008 ATI, les concurrents directs de NVIDIA, propose d'utiliser non pas un nouveau langage comme CUDA, mais une librairie de traitement parallèle connu de beaucoup, OpenCL. Elle devient compatible avec les cartes Radeon HD4000, cartes qui deviennent les premiers GPU de ATI[3] permettant le GPGPU. L'avantage est que Open CL est une librairie « open source », et ceux sachant programmer avec cette librairie (utilisée pour gérer le multitâche des processeurs standards) n'ont pas besoin de se former pour programmer son GPU.

De nombreuses études se sont penchées sur des études de viabilité du GPU. Quels sont ces avantages et ses inconvénients comparées aux CPU, est-ce qu'il est indispensable d'investir dedans ? Une étude pertinente a été réalisée par Ino et al. en 2005 (6). Ils se sont attardés sur une étude de performance d'une décomposition de LU, algorithme de base en algèbre linéaire. Ils soulignent ainsi plusieurs points importants : (i) L'importance du temps de transfert entre les mémoires CPU/GPU qui impacte la vitesse d'exécution. (ii) Le traitement des branches (IF, FOR etc…) est mieux géré par le CPU. (iii) Les résultats de la décomposition diffère car le traitement des valeurs « float » est différente sur le GPU. Ce sont ces études qui ont orientés NVIDIA dans son choix de programmation hétérogène avec CUDA et l'utilisation d'une mémoire unifiée GPU/CPU pour la prochaine architecture de cartes « Pascal » (en 2016). Lee et al. en 2010 (7) proposent une méthode de mesure de performance des GPU en les comparants aux CPU. L'originalité de l'article est d'analyser la température dégagée par les systèmes ce qui leur permet de conclure, à l'instar de Shen et al. (3) que les idées sur la trop grosse consommation des GPUs sont fausses. Plus récemment en 2015, Wu et al. (8) utilisent un algorithme d'apprentissage machine pour la mesure de performance des cartes graphiques. Bien que compliquée, la méthode apporte une mesure fiable, précise et donc efficace.

Un matériel est souvent cité lorsqu'il s'agit de comparer les GPUs, il s'agit des FPGA. Ceux-ci s'avèrent être plus modulables, plus puissant en terme de complexité algorithmique et plus économique en énergie et encombrement (ce qui prouve son intérêt pour les systèmes embarqués). Ils restent cependant moins puissants que les GPUs pour les tâches répétitives. Des études se sont penchées sur la fusion GPU/FPGA pour bénéficier des avantages de chacun, avec l'idée de remplacer le CPU par le FPGA. En 2014, Wu et al. proposent un système de ce type dans leurs travaux (9). Il en résulte que la combinaison permet une très bonne performance énergétique tout en alliant une grande capacité de calcul, ce qui leur permet de conclure que la fusion GPU/FPGA est plus performante que CPU/GPU.

[3] Aujourd'hui ATI a été absorbé par la société AMD, réputée comme étant le concurrent direct d'Intel dans le domaine des processeurs standards.

Aujourd'hui le « GPU computing » est reconnu dans l'industrie et est utilisé dans tous les domaines que ce soit en reconstruction tomodensitométrique 3D (10), en traitement d'image pour accélérer le « template matching » (11) ou encore plus récemment en ingénierie financière (12) par exemple.

Identification du problème

Suite à l'intérêt ce cette nouvelle manière d'optimiser les codes informatiques, nous avons choisi de nous attarder sur ce sujet. L'objectif final est de comprendre la programmation GPGPU pour pouvoir l'appliquer dans le projet de mémoire de Loïc dans le laboratoire LATIS. Il travaille sur du recalage échographique 3D où beaucoup de calculs sont nécessaires pour réussir à reconstruire un volume 3D à partir des images 2D récupérés par la sonde. Pour arriver à cette fin, plusieurs objectifs se dessinent :

1. La première étape est de comprendre le fonctionnement du GPU, pour cela nous étudierons nos deux cartes graphiques à disposition : NVIDIA GTX 560M et GTX 840M. Cette étape permettra de comprendre la programmation parallèle ainsi que les différents blocs matériels du système.
2. Ensuite nous nous initierons à la programmation hétérogène en nous aidant des nombreuses ressources en ligne. Le site officiel de NVIDIA propose des pages spécifiques au problème avec de nombreux tutoriaux et cours sur le sujet. Un premier programme de type « Hello World » nous permettra de valider l'installation.
3. Enfin, s'ensuivra une étude de performance basée sur deux applications : La première est une multiplication matricielle, la deuxième sera plus spécifique au domaine de la vision en étudiant les performances d'un filtrage de Sobel. Bien que cela semble simple dans un code dit « CPU », en GPU il y a de nombreux paramètres à prendre en compte pour optimiser les performances.

Pour arriver à nos fins, un étudiant doctorant de l'ETS spécialisé dans la programmation GPU nous propose son aide pour les diverses questions qui se poseront pendant le projet.

I - Le hardware du GPU

Avant de nous plonger dans l'informatique, il est nécessaire de clarifier ce qu'est un GPU (Graphic Processor Unit) en analysant le matériel. Nous allons dans cette partie commencer par poser les grandes différences de base entre un CPU et un GPU pour ensuite nous concentrer sur l'architecture précise de ces cartes. Enfin nous verrons les caractéristiques techniques des cartes employées durant ce projet.

I.1 - Comparaison avec le CPU

Les Unités centrales de traitement (CPU, Central Processing Unit) et les Processeurs graphiques (GPU, Graphics Processing Units) sont en fait constitués des mêmes éléments et ce sont leur architecture propre qui les différencie. On peut en effet résumer un processeur à quatre éléments :

- L'Unité Arithmétique et Logique (ALU, Arithmetic Logic Unit) permet au processeur d'effectuer des calculs. Les opérations de base comprennent l'addition, la soustraction, le changement de signe, les tests d'égalité, les opérateurs logiques, … Certaines ALU sont en mesures de manipuler des nombres à virgule flottante et ainsi d'effectuer des multiplications, des divisions ou des modulos.
- La mémoire vive dynamique (DRAM, Dynamic Random Memory Access) est le lieu où les données sont stockées pendant leur traitement. Elle peut être commune à plusieurs ALUs.
- La mémoire cache est une unité permettant l'accès plus rapide aux données de la DRAM. Les données y sont stockées temporairement uniquement.
- Enfin, une unité de contrôle va gérer les autres unités du processeur. Elle va initier les transferts de données et donner des calculs à exécuter aux ALUs.

L'image suivante schématise les architectures des GPU et des CPU. La principale différence et la plus significative est le nombre d'ALU. En effet, un CPU d'ordinateur sera constitué en général de quatre cœurs qui en fonctionnant en parallèle permettte les utilisations courantes d'un PC. Le GPU quant à lui comporte plusieurs centaines de cœur et présente donc un grand potentiel pour une application de parallélisme. Par ailleurs, la différence de taille entre une ALU de CPU et une ALU de GPU sur cette image a une signification. Les 4 cœurs du CPU seront en effet beaucoup plus performants, capables d'opérations complexes effectuées rapidement de manière séquentielle. Les cœurs du GPU n'offriront que des opérations de base et les exécuterons lentement. Ensuite, nous remarquons que les nombres d'unité de contrôle et de mémoire cache ont été démultipliés sur le GPU. C'est évidemment pour permettre aux nombreuses ALU de travailler en même temps, sans

attendre qu'une autre ait finie de recevoir ces instructions ou ses données. La DRAM reste la même dans les deux configurations, c'est un espace de stockage.

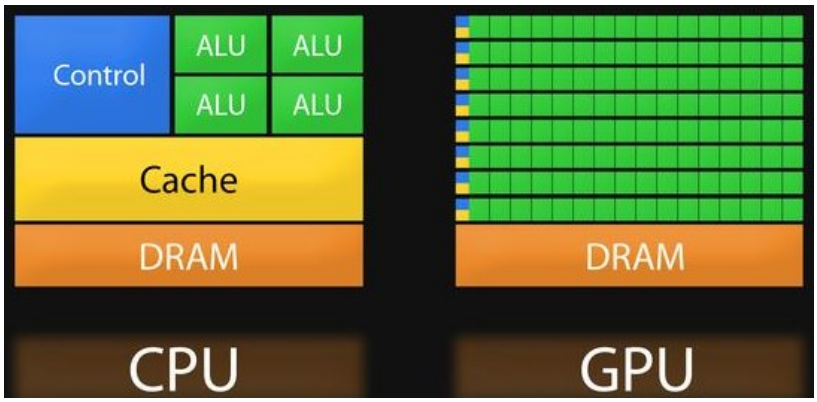


Figure 1 - Comparaison GPU/CPU

I.2 - L'architecture FERMI

Une des cartes que nous avons à disposition, la GTX 560M, dispose de l'architecture FERMI c'est pourquoi nous allons nous attarder sur celle-ci. Elle est basée sur l'architecture TESLA qui a permis la GPGPU et amène plusieurs évolutions majeures dans le domaine (en fonction du retour des utilisateurs des cartes TESLA) (13) :

- Tout d'abord l'amélioration des performances des calculs numériques « double précision » qui est essentiel dans les traitements numériques.
- Un support pour gérer plusieurs GPU en parallèle : Le contrôle de multiples cartes devient possible ce qui est un véritable bond notamment pour les serveurs ou laboratoires nécessitant de gigantesque puissance de calculs.
- Intégration des différents niveaux de cache mémoire à la manière des CPU.
- Augmentation de la mémoire partagée qui améliore la vitesse d'exécution des processus.
- Enfin, des améliorations sur l'intelligence du GPU (le GigaThread), l'horloge, le nombre de cœurs, les performances énergétiques.

Une carte graphique avec une architecture FERMI est composée de différents blocs matériels. Le premier élément est le Host Interface, c'est le bus du système qui va permettre aux différentes données de notre ordinateur de transiter. C'est par ce bus que pourra communiquer le CPU et en général correspond à une connexion de type PCI express. Ensuite il y a le GigaThread, qui est l'intelligence de la carte. Lorsque le CPU envoie des instructions à l'esclave (le GPU), le GigaThread va permettre de répartir les tâches entre les différentes unités de calcul : les cœurs.

Dans l'architecture FERMI (et les autres architectures qui suivront) il y a deux types de cœurs dans les processeurs graphiques. Le premier type est appelé SM (Streaming Multiprocessor) et est encadré en rouge sur le graphique, il correspond à une unité de calcul global où seront ensuite répartis différentes tâches selon les ordres du GigaThread. Le deuxième type, le cœur CUDA, correspond à l'unité de calcul de base est inclus dans les cœurs SM, on détaillera un cœur SM un peu plus tard.

Ensuite vient la mémoire et l'on en retrouve plusieurs types. On commence par la DRAM qui est la mémoire vive de la carte, toutes les données provenant du CPU arrive dans cet espace mémoire (une séquence vidéo composée de différentes images par exemple). Bien que la mémoire vive soit beaucoup plus rapide qu'une mémoire « morte » (disque dur) elle reste lente pour le transit de données dans les cartes. C'est pourquoi à la manière des CPU la mémoire est composé en différents niveau cache : La mémoire cache L2 est une mémoire plus rapide que la RAM partagée par les différents cœurs SM où se croisent les données que les cœurs SM se partage (une des images de la séquence vidéo). La mémoire cache L1 est plus spécifique aux cœurs SM et l'on retrouve les données que se partagent les cœurs CUDA (un patch de l'image contenu dans la mémoire cache L2).

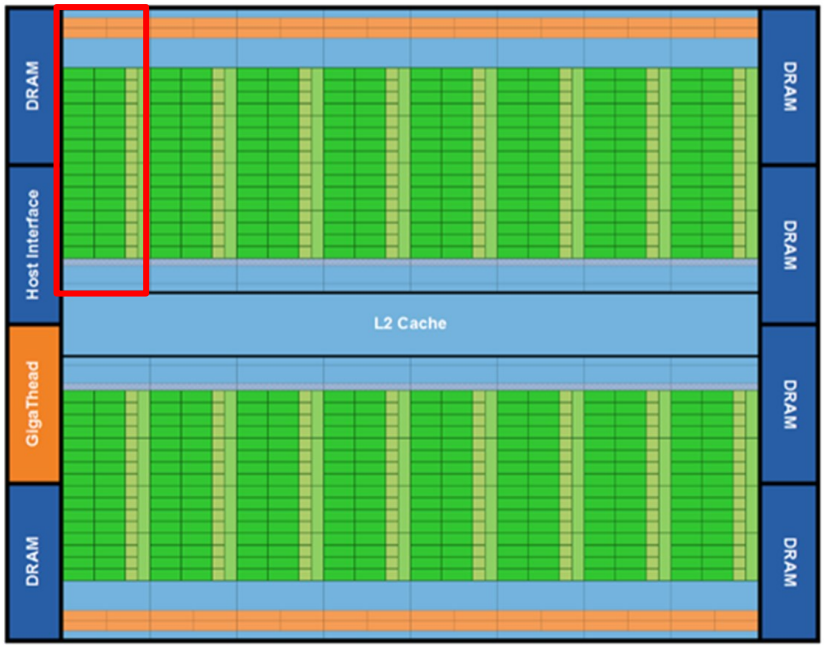


Figure 2 - Vision globale d'une carte graphique FERMI

Nous avons vu les différents blocs matériels de la carte graphique, voyons maintenant le contenu d'un cœur SM. Il se compose de différents espaces liés aux instructions stockés dans les registres (Instruction cache, Register File). Ces espaces sont gérés par le GigaThread par le biais des Warp Scheduler et Dispatch Unit qui permettent la répartition des instructions présentes et futures dans les cœurs CUDA. Un cœur SM est composé de 32 cœurs CUDA, un cœur SM peut donc faire 32 tâches basiques en parallèle (NVIDIA parle d'un « Warp »). Les SFU (Special Function Unit) permettent d'effectuer des opérations plus

complexe ($\sin(x)$, $\cos(x)$, $\sqrt{x}$, etc…). Enfin les unités LD/ST correspondent aux différentes opérations de transfert entre les mémoires.

Chaque SM dispose d'une mémoire cache L1 que se partagent les 32 cœurs CUDA. La mémoire cache L1 est gérée automatiquement par le processeur et il existe plusieurs méthodes pour optimiser les instructions à placer comme FIFO (First In First Out), LFU (Least Frequently Used) ou encore le plus populaire LRU (Least Recently Used). On retrouve aussi une mémoire partagée qui elle est à la volonté du programmeur, elle est gérée précisément dans le code (nous verrons l'utilité lors de la multiplication matricielle) et peut ne pas être utilisée.

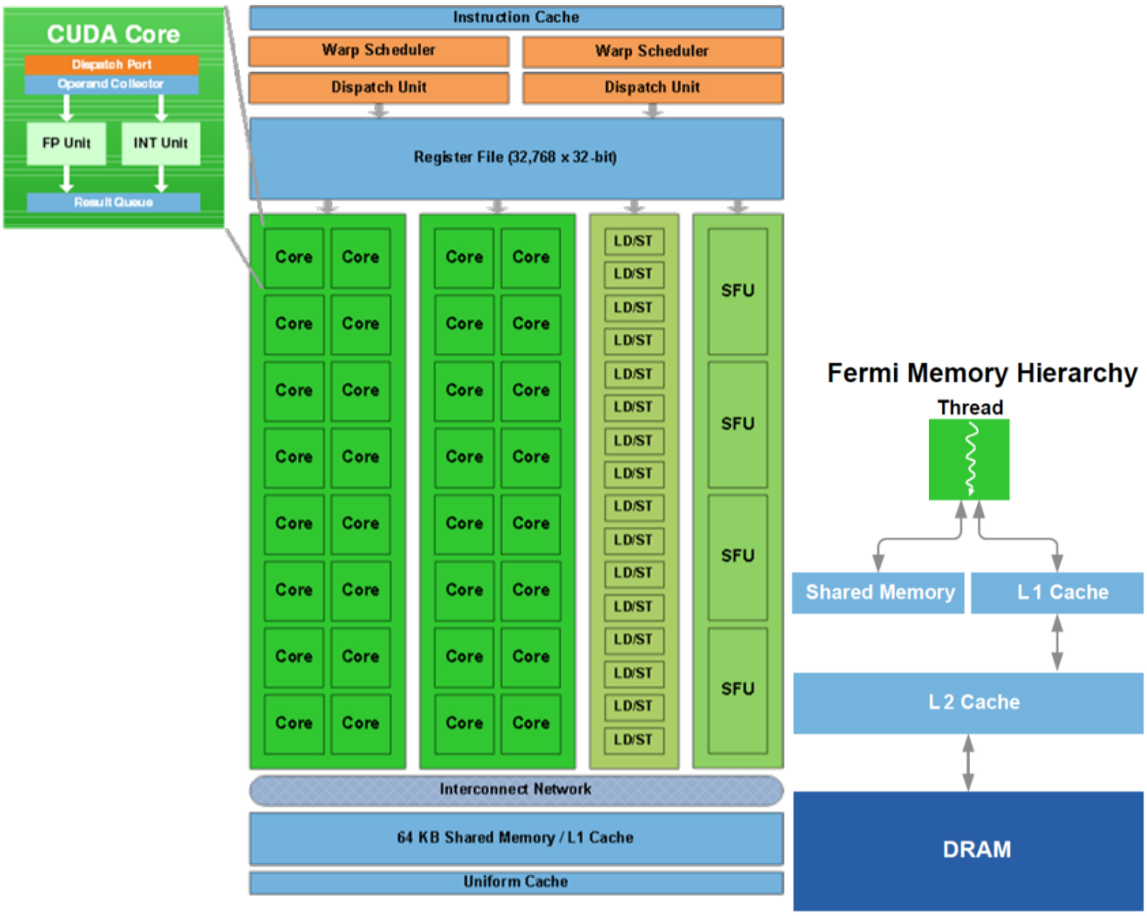


Figure 3 - Architecture d'un cœur SM et accès d'un cœur CUDA (thread ou tâche) aux différentes mémoires

Chacun de ces blocs est exécutée selon le cycle de l'horloge de la carte, ainsi en résumé par période d'horloge par cœur SM nous pouvons faire :

- 32 instructions basiques (addition, multiplication…)
- 16 opérations de transfert de mémoire
- 4 instructions complexes ($\sin(x)$, $\cos(x)$, $\sqrt{x}$, etc…)

Dû à la popularité et les performances de cette architecture, NVIDIA s'est basé sur ce modèle pour construire ses autres architectures.

I.3 - Spécifications technique des cartes utilisées

La première carte utilisée a été la NVIDIA GTX 560M, avec la lettre « M » correspond à la gamme portable des cartes. Elle a été annoncée en mai 2011 et possède l'architecture FERMI. Elle a la particularité de proposer OpenCL en plus de CUDA pour le GPGPU mais ne possède malheureusement pas de mémoire partagée. Elle été beaucoup utilisée dans des configurations d'ordinateur type « Gamer » ce qui explique la grande puissance de calcul à l'époque (14) :

Moteur GPU :

- 6 cœurs SM (32 cœurs CUDA)
- Horloge du processeur de 1550 Mhz
- 24.8 Gp/s (nombre de pixels traités par seconde)

Mémoire :

- 1536 Mo de mémoire vive GDDR5
- Vitesse mémoire de 1250 Mhz
- Taux de transfert de 60 Go/s
- 192 bits de taille de bus

Caractéristiques :

- OpenGL 4.5
- OpenCL 1.1
- DirectX 11
- 3D (Blu-Ray, Gaming, Photos)

La seconde carte graphique est plus récente, la NVIDIA GeForce 840M. Elle possède la dernière architecture en date, Maxwell, et a été introduite en 2014. La nouveauté se situe dans les performances de ces différents modules, mémoire plus rapide, SM contenant un tiers en moins de cœurs par rapport à l'architecture FERMI pour 90% de performance (15):

Moteur GPU :

- 3 cœurs SM (128 cœurs CUDA chacun)
- Horloge du processeur de 1550 Mhz

Mémoire :

- 4096 Mo de mémoire vive DDR3
- Vitesse mémoire de 2000 Mhz

- 64 bits de taille de bus

Caractéristiques :

- OpenGL 4.4
- OpenCL 1.1
- DirectX 12 API
- 3D (Blu-Ray, Gaming, Photos)

II - La programmation du GPU

Maintenant que les bases matérielles sont clarifiées, nous pouvons nous plonger plus précisément dans la programmation du GPU. Nous verrons les différentes façons de programmer avec un GPU de NVIDIA, et nous nous attarderons sur chacune de ces façons pour expliquer la théorie.

CUDA est en fait une plate-forme qui propose 3 manières de programmer avec un GPU NVIDIA : nous commencerons par l'utilisation de OpenACC, nous verrons par la suite les différentes librairies disponibles fonctionnant sous CUDA et nous finirons par la programmation de kernels spécifiques en C/C++.

II.1 - La programmation avec OpenACC

OpenACC est un ensemble de directives (appelées directives de préprocesseurs) envoyés au processeur pendant la compilation du programme. À la manière de OpenMP, on écrit dans la structure du programme des « indices » au compilateur. Ce qui fait l'intérêt de cette méthode est qu'elle ne nécessite aucune connaissance en programmation de carte graphique. On peut utiliser cette manière de programmer sur n'importe quel code « CPU » (en langage C++ par exemple) en rajoutant quelques lignes dans le code, et en ayant bien sûr installé CUDA au préalable.

```
#include <stdio.h>
  #define N 1000000
  int main(void) {
  double pi = 0.0f; long i;
  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++) {
     double t= (double)((i+0.5)/N);
     pi +=4.0/(1.0+t*t);
  }
  printf("pi=%16.15f\n",pi/N);
  return 0; }
```

Lorsque le compilateur voit les directives **`#pragma`** il comprend que cette boucle devra être utilisé par le GPU ce qui augmentera significativement les performances. OpenACC est souvent couplé à une analyse de type « profiler » qui permet de pointer les endroits d'un code les plus gourmands en termes de vitesse d'exécution. Il est souvent utilisé en industrie pour sa facilité d'intégration, mais constitue la manière la moins optimisée d'utiliser les performances de sa carte graphique.

II.2 - La programmation avec des librairies supportant CUDA

La deuxième manière de programmer est d'utiliser des bibliothèques proposant des fonctions préprogrammées, à la manière de standards de l'industrie comme OpenCV ou OpenCL… On retrouve beaucoup de bibliothèques et nous en allons lister celles que propose CUDA lors de l'installation (approuvés par les programmeurs de NVIDIA) :

- Math : Il s'agit en fait de la librairie « math.h » mais optimisée pour le GPU. On retrouve ainsi toutes les opérations que propose « math.h » telles que les fonctions trigonométriques, hyperboliques, exponentielles… Ces différentes opérations exploitent les unités de calculs SFU avec des développements de Taylor par exemple.
- cuFFT : C'est une librairie spécialisé dans le traitement de Fourier, opération de base en traitement de données. Elle propose d'effectuer des transformées complexes ou réelles sur des données 1D/2D/3D avec différents formats (simple, ou double précision).
- cuBLAS : BLAS pour « Basic Linear Algebra Subroutines » propose des opérations sur différents types de donénes (réel, complexes…) pour des inversions de matrice, des décompositions LU. Cette librairie est donc surtout utilisé pour de l'algèbre linéaire.
- Thrust : Thrust est une bibliothèque spécialisée dans le traitement de données parallèle. Il propose de trier des tableaux, des matrices et est beaucoup utilisée en optimisation combinatoire.
- cuSPARSE : Cette bibliothèque ressemble à cuBLAS mais se spécialise dans les matrices dites « creuses ». Ces matrices sont beaucoup utilisées en algèbre car elles permettent d'économiser la place en mémoire, en effet les valeurs nulles « 0 » ne sont plus stockés en mémoire.
- cuRAND : C'est la librairie à utiliser lorsque l'on veut générer des nombres aléatoires. C'est une problématique très importante dans le domaine de la cryptographie ou de simulations diverses par exemple. On veut en fait générer des nombres complètement aléatoires et donc qui ne soient pas « prédictibles ».
- NPP : Enfin, la dernière bibliothèque proposée dans le package de NVIDIA est NPP qui inclue de nombreuses fonctions de traitements d'images. Cette bibliothèque sera un peu plus détaillée par la suite.

Chaque bibliothèque à sa particularité d'utilisation mais la philosophie générale est la même : après avoir installé les librairies, nous pouvons utiliser les fonctions qu'elle propose. Cette méthode de programmation est la plus intéressante car elle combine la facilité d'utilisation (avec des fonctions déjà programmées) et la rapidité d'exécution (fonctions optimisées par des professionnels).

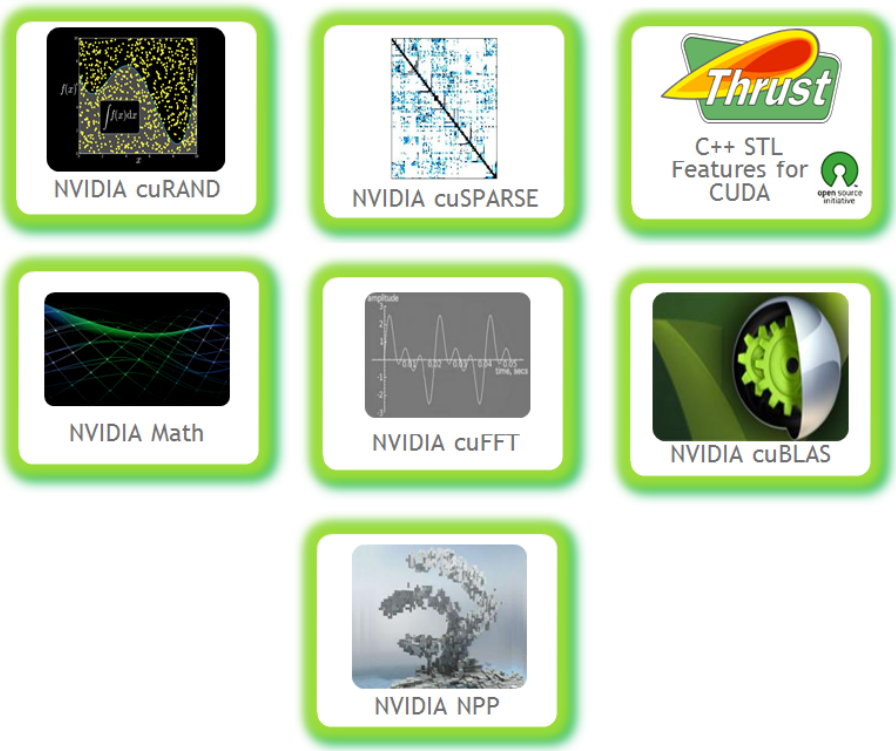


Figure 4 - Les différentes librairies que propose CUDA lors de son installation

II.3 - La programmation avec CUDA C/C++

CUDA, c'est une plateforme pour la programmation parallèle. Elle donne accès à la programmation hétérogène à travers des interfaces permettant de coder dans des langages courants comme Java, Python ou dans le cas qui nous concerne, en C/C++. L'approche logicielle de la programmation parallèle par CUDA ne correspond pas exactement à ce qu'il se passe au niveau matériel. La plateforme est une boite noire masquant la distribution des tâches dans les différents cœurs et SM et dans le temps. Nous allons donc découvrir ici les notions qui rendent possible la programmation CUDA et comment les utiliser.

Tout d'abord, une « fonction » de GPU est appelée « kernel » (noyau). Le kernel sera alors exécuté sur le GPU par un groupe de threads travaillant en parallèle les uns des autres. Un thread ne peut exécuter qu'une opération simple comme une addition, une multiplication, … L'application GPU basique demande d'effectuer plusieurs fois le même

calcul et les threads permettent de faire tous ou une partie de ces calculs en même temps. Les threads sont groupés par block. Sous l'architecture Maxwell, nous pouvons créer des blocks contenant jusqu'à 1024 threads. Les blocks forment quant à eux une « grid » (grille) qui contiendra un nombre de blocks pouvant atteindre $(2^{31} - 1)$ tant que le nombre de threads total ne dépasse pas cette même limite. Le CPU reste le maître (host) de l'application de GPU computing, il décide quel kernel doit être lancé. Il décide aussi les dimensions des grid (une grille par kernel) et des blocks associés et alloue également la mémoire sur le GPU, son esclave (device). L'organisation logicielle CUDA est illustrée par le schéma suivant. Les blocks et threads pouvant être distribués respectivement en grid et blocks à 3 dimensions (seulement deux présentées sur l'image):

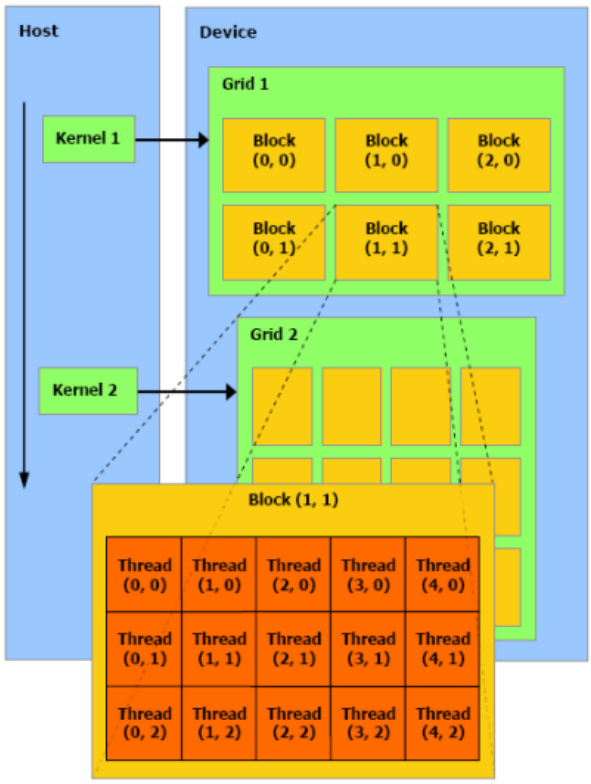


Figure 5 - Kernel, grid et block

Nous verrons que la gestion de la mémoire doit faire l'objet d'une attention toute particulière en GPU computing. Le schéma suivant nous montre les différentes mémoires du processeur graphique. Nous ne détaillerons que les plus importantes pour notre projet :

- La mémoire globale est la mémoire de et vers laquelle seront reçus et envoyés les données avec le CPU. Elle est commune à toute la grid du kernel.
- Les registres sont utilisés par les threads pour stocker leurs résultats de calculs intermédiaires.
- Les threads possèdent chacun une mémoire locale dans le cas où pas assez de registres sont disponibles ou trop de données sont à stocker pour les registres.
- La mémoire partagée (shared memory) est une mémoire rapide d'accès et partagée entre les threads d'un même bloc. C'est à l'utilisateur de décider de l'utiliser ou non.

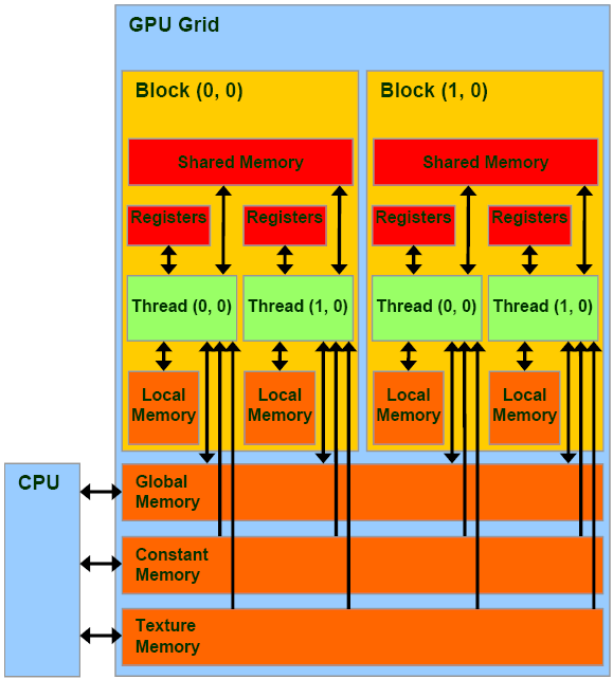


Figure 6 - Contenu d'une "grid"

Avant d'exécuter des opérations sur le processeur graphique faut donc écrire le code qui permettra au CPU de gérer l'application GPU. La première chose à gérer est comme pour toute application la mémoire. Le CPU va demander au GPU d'allouer une zone de sa mémoire d'une certaine taille pour recevoir des données ou stocker ses résultats. Nous le faisons en utilisant la fonction :

```
type *variable_GPU = 0;
cudaMalloc((void**)&_variable_GPU, size_variable*sizeof(type));
```

Ces zones de mémoires allouées devront être libérées une fois qu'elles n'auront plus d'utilité par la fonction :

```
cudaFree(variable_GPU);
```

Nous pouvons alors envoyer les matrices à multiplier dans la mémoire du GPU ou recevoir la matrice résultante grâce à la fonction suivante. Nous utilisons les constantes `cudaMemcpyHostToDevice` et `cudaMemcpyDeviceToHost` pour définir le sens du transfert.

```
cudaMemcpy(destination, source, size, cudaMemcpyHostToDevice);
```

Pour lancer la partie du programme à exécuter sur le GPU, nous appelons le ou les kernels. Comme nous l'avons vu, chaque kernel devra être appelé en lui associant le nombre de blocks et le nombre de threads par blocks à utiliser. Le qualificatif __global__ signifie que la fonction qui suit doit être exécutée par le GPU :

```
__global__ Kernel () {}
Kernel << <nbBlock, nbThreadperBlock >> >();
```

Enfin, des fonctions CUDA ne sont pas nécessaires à l'exécution du programme mais permettront d'une part à déterminer les sources d'erreurs et d'autre part le bon déroulement de l'analyse de performance. Les quatre fonctions utilisées sont référencées dans le tableau suivant :

Fonctions	Appel	Commentaire
`cudaSetDevice()`	Au début du programme	Sélectionne le GPU à utiliser
`cudaDeviceReset()`	A la sortie du programme	Permet d'analyser l'intégralité du programme
`cudaGetLastError()`	Après appel kernel	Informe si le kernel s'est bien lancé
`cudaDeviceSynchronize()`	Après appel kenerl	Informe si le kernel a fini de d'exécuter et si des erreurs sont survenues

III - Etude et performance du GPGPU

L'étude de performance est une étape importante, car c'est elle qui va permettre la validation du projet. Nous commencerons par expliquer les différents outils que nous avons utilisés pour réaliser cette étude, pour ensuite nous attarder sur la multiplication matricielle et un filtrage de Sobel.

III.1 - Les différents outils utilisés

Beaucoup d'outils ont été utilisés pour mener à bien ce projet. Tout d'abord le logiciel Visual Studio 2013 professionnel a permis de faire la programmation. Il propose des options avec CUDA qui facilite la programmation sur GPU (notamment des options pour créer des projets dédié à CUDA). Toute la boite à outil pour CUDA est disponible à l'adresse suivante : https://developer.nvidia.com/cuda-downloads.

Le dossier se compose de divers éléments notamment les différentes librairies dont nous avons fait référence précédemment. Il comporte bien évidemment le compilateur NVIDIA NVCC pour coder en C/C++. Nous avons aussi utilisé la librairie OpenCV 2.4.1 pour afficher et gérer les images : http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.10/opencv-2.4.10.exe/download.

Suite à la complexité pour intégrer la librairie dans un projet, il existe des tutoriaux dans la documentation :http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html#windows-installation (Windows).

Ainsi qu'une vidéo d'explication pour pouvoir compiler son premier programme avec OpenCV sur Visual 2013 : https://www.youtube.com/watch?v=vwhTKsvHwfQ

La librairie NVIDIA NPP est disponible lors de l'installation de CUDA et ne nécessite pas de réglages à proprement parler.

Pour ce qui est de l'analyse de performance nous avons utilisé trois outils différents :

- Le premier (pour analyser les temps d'exécution des programmes « CPU ») est un analyseur de performance offert par Visual Studio 2013 (onglet « Analyser » / « Performances et Diagnostics »)
- Le deuxième est NSIGHT qui propose un environnement dédié à l'analyse des fonctions CUDA (spécifiques au GPU) disponible sous : http://www.nvidia.com/object/nsight.html

- Enfin, le dernier outil est facilement trouvable sur internet : HW Monitor récupère diverses informations des capteurs de l'ordinateur (température, voltage, puissance).

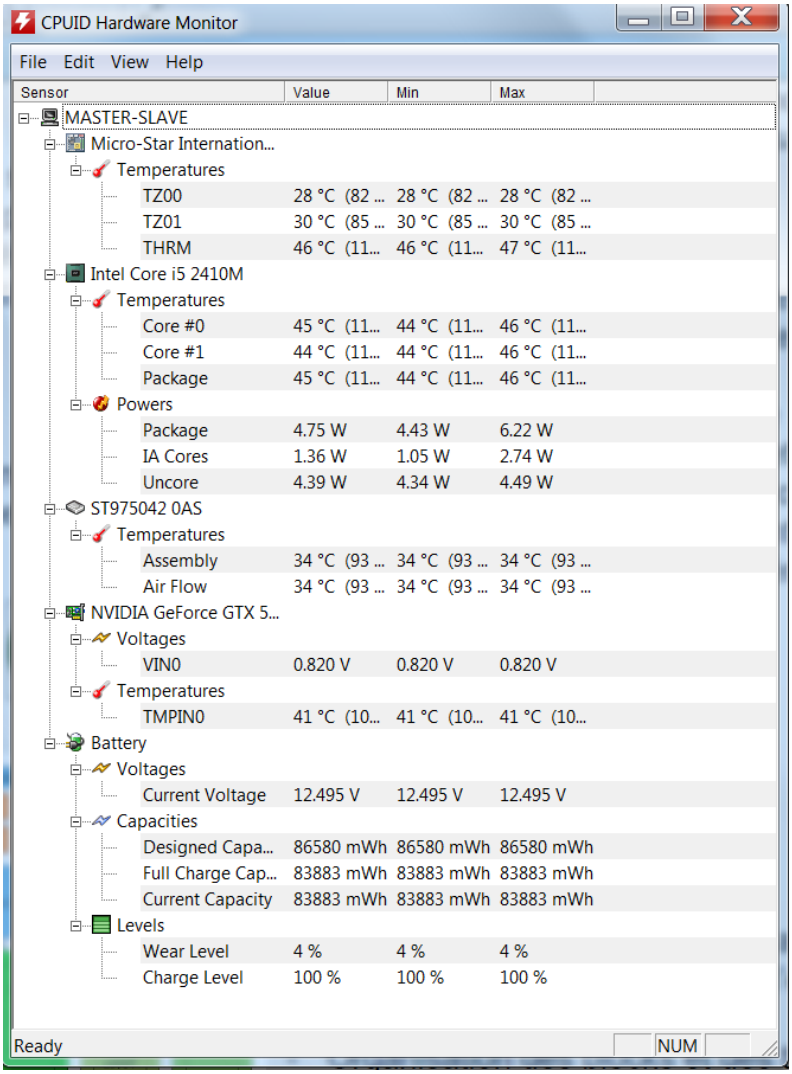


Figure 7 - Fenêtre de HW Monitor

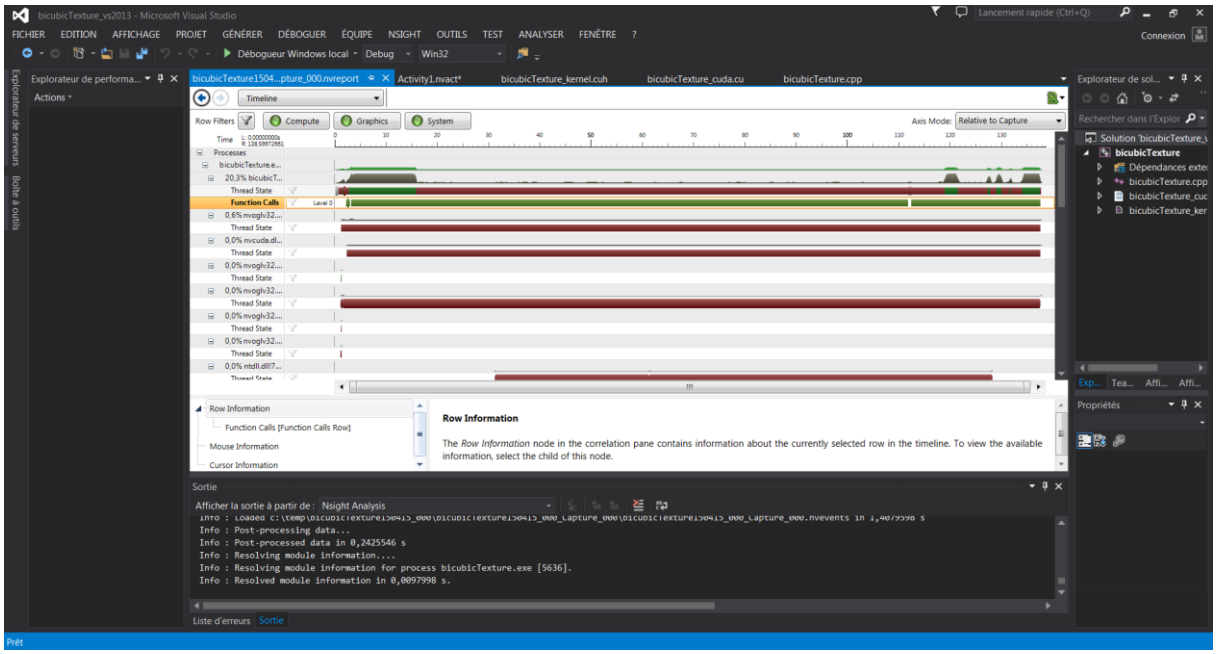


Figure 8 - Visual Studio 2013 Professional et NVIDIA Nsight Visual Studio Edition

III.2 - Multiplication matricielle avec CUDA

Nous avons choisi comme premier cas d'étude de performance la multiplication matricielle pour plusieurs raisons. En premier lieu, c'est un exemple fondamental de la programmation CUDA et du parallélisme en générale. Elle fait appel aux principes de base mais peut également être appliquée en utilisant des techniques plus avancée. Il est aussi facile pour nous de valider les bonnes opérations du GPU à travers une application simple comme celle-ci, il nous suffit de comparer le résultat obtenu avec le résultat du calcul effectuer sur CPU. Mais la multiplication matricielle nous permettait également d'effectuer des calculs sur les éléments de tableaux à deux dimensions et ainsi ne pas trop nous éloigner du concept d'image. Nous pourrons agir sur les dimensions des matrices comme nous le ferions pour travailler sur des images de résolution plus ou moins grande.

Nous allons reprendre ici les étapes qui nous ont permis de découvrir et appliquer les notions de programmation CUDA et tendre vers une multiplication de matrice optimale dans son exécution et la gestion des ressources. Nous utiliserons les outils d'analyse de performances pour observer l'accélération du calcul par le GPU par rapport au CPU. La carte utilisée tout au long de cette partie est la GEFORCE 840M. Comme il est courant de le faire, nos matrices seront représentées par des tableaux à une seule dimension, les lignes étant stockées à la suite les unes des autres dans le tableau. Nous n'utiliserons également que des matrices carrés de largeur « width ». Ainsi, pour accéder à l'élément de la $i^{ème}$ ligne et de la $j^{ème}$ colonne d'une matrice M, il faudra appeler :

$$M[i * width + j]$$

Etape 1 : premier kernel

Cette première tentative n'utilisera qu'un seul block et nous donnera un premier aperçu du langage utilisé pour programmer en parallèle. Chaque thread sera associé à un élément de la matrice à calculer et donc chaque thread effectuera la même opération. Le bloc créé est en deux dimensions, ainsi, les threads pourront être considérés comme les éléments d'une matrice. Pour faire référence aux threads nous utilisons leurs coordonnées stockées dans la structure $threadIDx$:

$$threadIdx.x = n° \, colonne$$
$$threadIdy.y = n° \, ligne$$

Le code suivant parcours les éléments des matrices A et B pour remplir la matrice résultante M. La figure associée illustre un des calculs effectués mais il faut garder en tête que cette opération est exécutée par tous les threads en même temps, chacun pour un élément différent de la matrice M.

```
__global__ void matriXKernel(float *A, float *B, float *M, unsigned int width)
{
	unsigned int row = threadIdx.y,
		     col = threadIdx.x;

	float Mvalue = 0;

	for (int k = 0; k < width; k++)
		Mvalue += A[row*width + k] * B[k*width + col];

	M[row*width + col] = Mvalue;
}
```

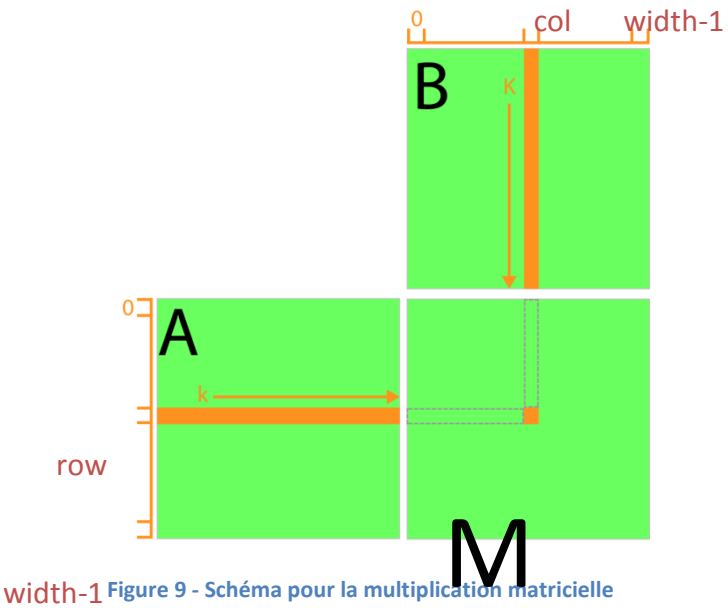


Figure 9 - Schéma pour la multiplication matricielle

Tous les identifiants de threads étant concernés dans ce code, il est courant d'ajouter des conditions assurant la limitation des valeurs de *row* et de *col* aux coordonnées voulues :

```
for (int k = 0; k < width; k++)
              if (row < width && col < width)
                     Mvalue += A[row*width + k] * B[k*width + col];
if (row < width && col < width)
       M[row*width + col] = Mvalue;
```

Cette application est fonctionnelle, toutefois, un bloc ne peut contenir qu'un maximum de 1024 (32^2) threads sur l'architecture Maxwell. De plus, un block ne peut être exécuté que sur un seul et même Streaming Multiprocessor (SM) et il est préférable pour une utilisation plus rentable du GPU d'occuper tous les SM. C'est pourquoi l'organisation en block est utile et indispensable pour le traitement de nombreuses données.

Etape 2 : Blocks et threads

Introduire les blocks dans le programme est très simple, il suffit d'utiliser les identifiants $blockIdx$ et dimensions $blockDim$ de ceux-ci pour faire référence aux lignes et aux colonnes :

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
```

La problématique est ici dans le choix de ces dimensions. En effet, la configuration maxwell nous impose un maximum de $32\ x\ 32$ threads par block mais quelle est quelle est la configuration optimale ? Il n'est pas toujours aisé de faire le lien entre l'organisation logiciel du parallélisme (threads, blocks, …) et son exécution matérielle (cœurs, SM, …). Nous avons donc mesuré les temps d'exécution de la multiplication de matrice en fonction du nombre de threads par blocks. Comme la figure suivante nous le montre pour les cas de matrices $32x32$ et $64x64$, un trop grand nombre de blocks ralentit grandement le temps de calcul. D'une autre part, la saturation du nombre de thread/block semble ralentir l'application mais dans une moindre mesure.

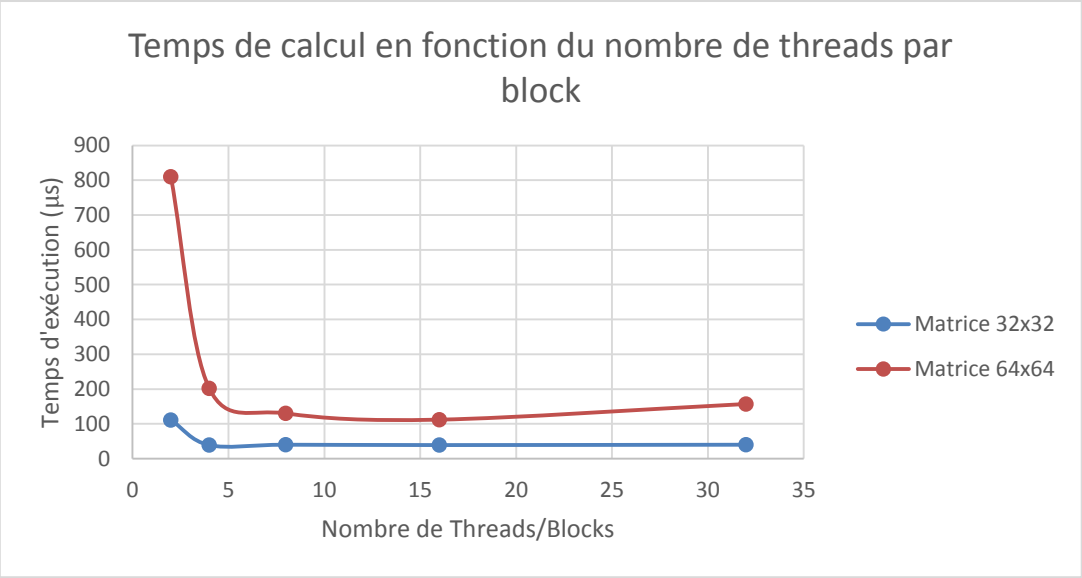


Figure 10 - Temps d'exécution de la multiplication matricielle en fonction des threads par bloc

L'utilisation des blocks permet de traiter des matrices de grandes dimensions, ce code est la version la plus simple à mettre en place d'un programme GPU effectuant une multiplication matricielle, mais d'autres techniques CUDA existent.

Etape 3 : Mémoire partagée

Une des caractéristiques les plus utiles de la programmation CUDA est la mémoire partagée qui se doit d'être utilisée lorsqu'elle est possible. Nous le rappelons, la mémoire partagée permet aux threads d'un même block d'avoir accès très rapidement à des données communes.

La mémoire partagée nécessite la copie des données de la mémoire globale à la mémoire partagée de chaque bloc. L'objectif est de minimiser le nombre d'appel à la mémoire globale, lente. En effet, dans les codes précédents, chaque thread demande les données dont il a besoin à la mémoire, ce qui implique que les mêmes données seront chargées plusieurs fois pour différents threads. Grâce à la mémoire partagée, nous pouvons charger les données qui sont communes aux threads d'un même block une seule fois et les utilisées de manière très rapide. On déclare ces données partagées à l'aide du qualificatif $\_\_shared\_\_$.

Le code suivant présente comment la mémoire partage est mise en place pour la multiplication matricielle. En fait, le calcul de chaque élément se fera en plusieurs fois au fur et à mesure que la mémoire partagée des différents blocks seront chargées. Cette technique nécessite l'utilisation d'une nouvelle fonction : $\_\_syncthreads()$ pour que les threads partageant des données soient synchrones. Encore une fois, le GPU est une vraie boîte noire et il est difficile d'imaginer comment et dans quel ordre les actions y sont effectuées. Donc

lorsque des threads partagent des données, il est important de s'assurer que tous ont terminées leur chargement ou calcul avant d'utiliser le résultat de leur collaboration.

```
#define TILE_WIDTH 32      // largeur block

__global__ void matriXKernel_shared(float *A, float *B, float *M, unsigned int width)
{
	int	bx = blockIdx.x, by = blockIdx.y,
		tx = threadIdx.x, ty = threadIdx.y,
		row = by*TILE_WIDTH + ty, col = bx*TILE_WIDTH + tx;

	float Mvalue = 0;

	// Pour chaque bloc
	for (int m = 0; m < (width - 1) / TILE_WIDTH + 1; m++)
	{
		// Charger la mémoire partagée
		__shared__ float	As[TILE_WIDTH][TILE_WIDTH],
					Bs[TILE_WIDTH][TILE_WIDTH];

		if (row < width && (m*TILE_WIDTH + tx) < width)
			As[ty][tx] = A[row*width + (m*TILE_WIDTH + tx)];

		if ((m*TILE_WIDTH + ty) < width && col < width)
			Bs[ty][tx] = B[(m*TILE_WIDTH + ty)*width + col];
		// Attendre que toutes les données soient chargées
		__syncthreads();

		// Multiplication matricielle avec les données chargées
		for (int k = 0; k < TILE_WIDTH; k++)
			Mvalue += As[ty][k] * Bs[k][tx];
		// Attendre la fin du calcul
		__syncthreads();
	}

	// Remplir la matrice M
	if (row < width && col < width)
		M[row*width + col] = Mvalue;
}
```

Le choix du nombre de threads par block est encore à considérer mais cette fois la décision est plus simple. En effet, les données ne sont partagées qu'à l'intérieur des blocks et nous voulons partager un maximum de données. Nous choisirons donc d'utiliser les blocs les plus grands possibles, soient $32x32$ threads.

Analyse de performances

Nous disposons alors de deux programmes hétérogènes capables de multiplier deux matrices. A l'aide de quelques boucles, nous codons cette multiplication sur CPU pour pouvoir comparer son temps à celui de l'exécution des kernels. La figure suivante présente les mesures effectuées pour des matrices de taille $4x4$ à $2048x2048$ avec des axes à l'échelle logarithmique pour plus lisibilité. Nous remarquons que si le CPU est plus de 10 fois plus rapide pour la matrice $4x4$, les capacités en parallélisme du GPU prennent rapidement

l'avantage et accélère l'application jusqu'à prendre 60 fois moins de temps. Par ailleurs, nous notons des performances supérieures de la part du programme utilisant la mémoire partagée mais son avantage est moins net lorsque les matrices augmentent en taille. Ceci s'explique du fait que plus les matrices grandissent, plus le nombre de blocks augmente et donc plus la mémoire globale est sollicitée, les blocks ayant atteint leur taille maximale.

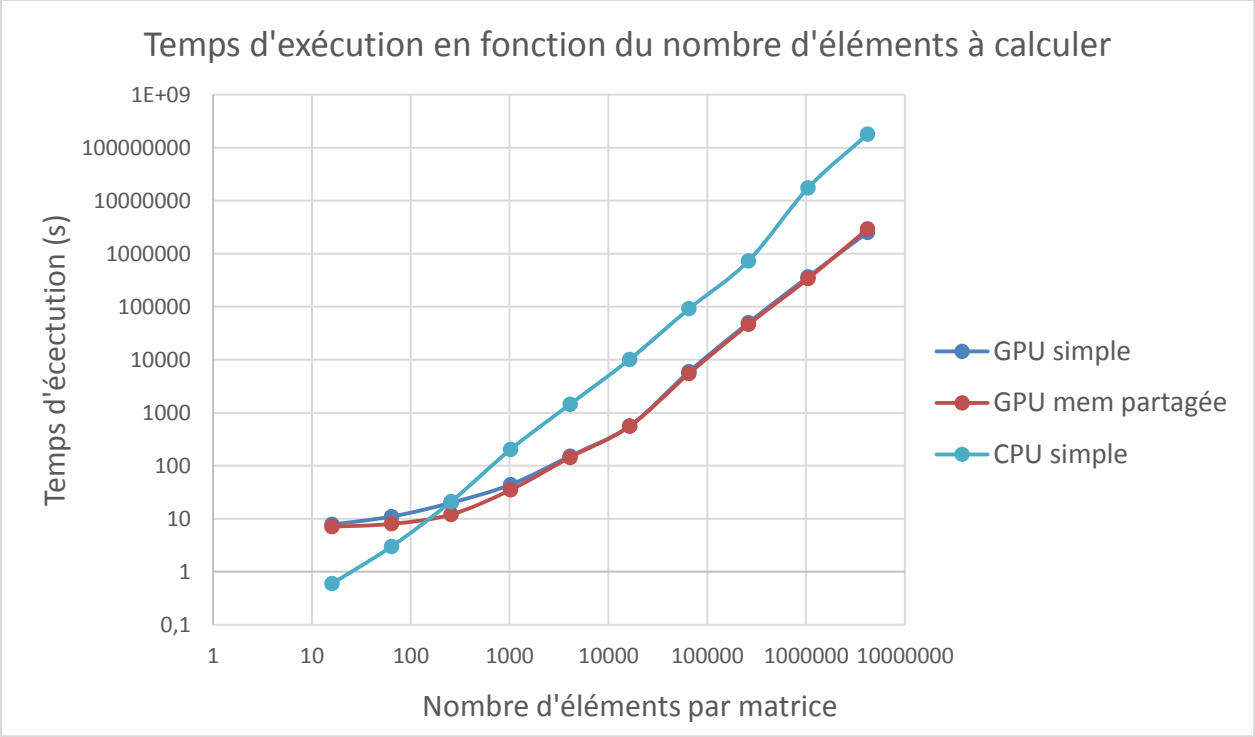


Figure 11 - Temps d'exécution de la multiplication matricielle en fonction du nombre d'éléments

Les échelles logarithmiques empêchent de bien visualiser le gain de temps. Le graphique suivant présente donc le pourcentage d'accélération offert par la carte graphique. Le GPU surclasse rapidement le CPU et accélère ensuite en se rapprochant doucement des 99% de diminution du temps.

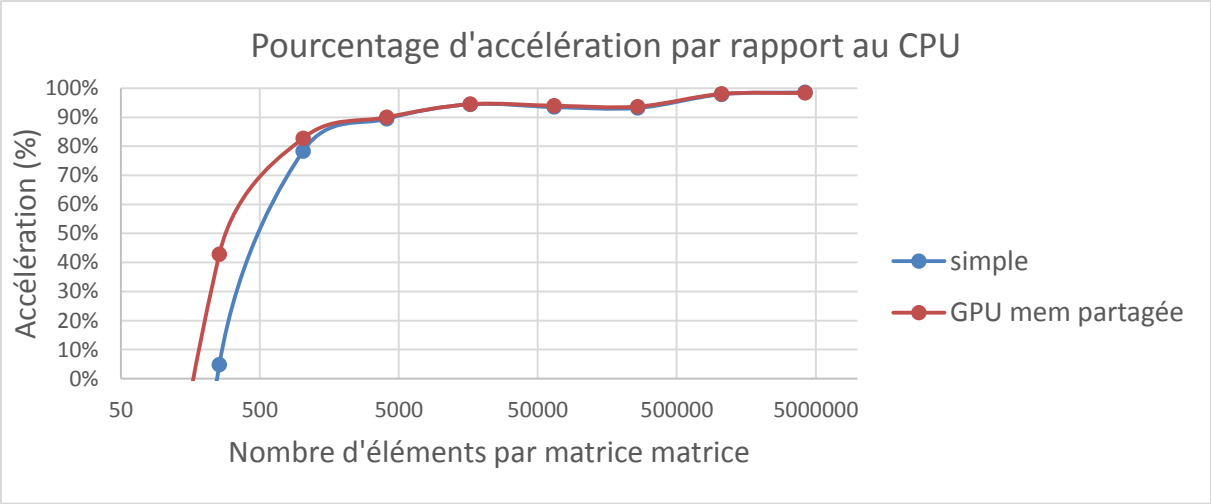


Figure 12 - Gain sur l'exécution du GPU par rapport au CPU

L'utilisation de GPU pour ce genre de tâche est alors évidente. Même en considérant le temps nécessaire au transfert des données et au contrôle du GPU par le CPU, qui est quasiment négligeable devant l'accélération du traitement de grandes quantités de données, le CPU peut encore gagner du temps en mettant des efforts dans d'autres applications en attendant les résultats du GPU. Nous allons donc par la suite tester ces performances dans un cadre plus proche de la vision par ordinateur, mais avant revenons sur les difficultés rencontrés et une remise en question de notre analyse.

Difficultés rencontrées

Au-delà de l'installation parfois fastidieuse des logiciels, analyseurs de performances et librairies, ce sont les chemins d'accès aux fonctions CUDA qui ont dans un premier temps posés problème. En effet, suivant le matériel et les versions utilisés, les méthodes utilisées dans les tutoriels et proposées dans les forums ne fonctionnent pas toujours, en particulier pour inclure la fonction $\_\_synchthreads()$. Si vous rencontré vous-même ce genre de problème, voici la méthode et les fichiers que nous avons inclus :

```
#ifndef __CUDACC__
#define __CUDACC__
#endif

#include <stdlib.h>
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <device_functions.h>
```

Concernant les analyses de performances, une difficulté résidait dans le fait que la gestion des ressources du CPU et du GPU n'est pas visible. Différentes séries de mesures peuvent donner des résultats très différents dépendamment des activités en cours sur l'ordinateur. De manière générale, si les résultats changeaient, la comparaison des performances restaient la même. Le GPU est toujours plus rapide que le CPU par contre la mémoire partagée n'a pas toujours donner les meilleurs résultats. En effet, les temps d'exécution étant très proche de ceux d'un code GPU simple, il est possible que les variations d'une mesure à l'autre échange l'ordre de l'application la plus rapide. Nous pouvons souligner que le code GPU est assez simple et il est donc difficile de l'accélérer parce que rajouter du code rajoute du temps à l'exécution.

Ensuite, une question nous a traversés l'esprit, puisque le GPU s'occupe normalement de l'affichage de l'ordinateur, est-ce que l'utilisation de la carte graphique pour une autre application ne perturbe pas le dit affichage ? Il est arrivé que lors de long calcul que le système d'exploitation nous avertisse que le périphérique d'affichage avait été perdu le temps d'un instant. Toutefois ceci est dû à une erreur lors de l'allocation de ressources à l'activité demandée et ce n'est pas un conflit d'affichage qui est à l'origine de l'erreur. En fait, c'est le plus souvent le CPU (par le biais du chipset graphique intégré) qui s'occupe de l'affichage et requiert l'aide du GPU pour la 3D ou les applications plus complexes.

Enfin, l'utilisation de deux analyseurs de performances différents pour une comparaison n'est pas la meilleure des méthodes, leurs résultats n'étant surement pas obtenus par le même procédé. De plus, il a fallu exécuter le programme pour chaque analyseur, ce n'est donc surement pas le même temps réel qui est mesuré. Cependant, les nombreuses séries de mesures présentaient une tendance commune, l'écart entre les temps GPU et CPU ne laisse pas d'ambiguïté et les résultats obtenus correspondent à ce que la littérature nous avait fait espérer.

III.3 - Filtrage de Sobel horizontal

Nous avons créée deux projets différents pour analyser les performances d'un filtrage de Sobel sur la carte GTX 560M :

- Le premier est un projet CUDA utilisant la librairie NVIDIA NPP
- Le deuxième est un filtrage à l'aide de la bibliothèque OpenCV

Pour cette étude, nous avons choisi d'utiliser des images de la fameuse « Lena » au format portable graymap (pgm). C'est un format simple d'utilisation et qui ne pèse pas beaucoup. Dans notre programme, les pixels sont codés sur 8 bits non signés (de 0 à 255) pour simplifier le traitement et le temps de calcul. Le noyau utilisé pour le filtrage est de taille 3 avec la dérivée selon y (pour détecter les contours horizontaux) :

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Figure 13 - "Lena.pgm"

Commençons par expliquer le filtrage par GPU avec NVIDIA NPP. Nous allons nous attarder sur le cœur du programme, toutes les routines d'erreurs et de vérification ne seront pas expliquées (le fichier en entier est disponible en annexe) :

```
sFilename = "Lena.pgm"
	npp::ImageCPU_8u_C1 oHostSrc;
	npp::loadImage(sFilename, oHostSrc);
	npp::ImageNPP_8u_C1 oDeviceSrc(oHostSrc);
	NppiSize oMaskSize = { 3, 3 };
NppiSize oSizeROI = { (int)oDeviceSrc.width() - oMaskSize.width + 1,
(int)oDeviceSrc.height() - oMaskSize.height + 1 };
	npp::ImageNPP_8u_C1 oDeviceDst(oSizeROI.width, oSizeROI.height);
		clock_t begin = clock();
		int count = 10000;
		for (unsigned int i = 0; i < count; i++)
		{
			NPP_CHECK_NPP(
				nppiFilterSobelHoriz_8u_C1R(oDeviceSrc.data(),
				oDeviceSrc.pitch(),
				oDeviceDst.data(), oDeviceDst.pitch(),
				oSizeROI));
		}
		clock_t end = clock();
		float diffms = ((end - begin) * 1000) / CLOCKS_PER_SEC;
		diffms = diffms / count;
	npp::ImageCPU_8u_C1 oHostDst(oDeviceDst.size());
	oDeviceDst.copyTo(oHostDst.data(), oHostDst.pitch());
	saveImage(sResultFilename, oHostDst);
	std::cout << "Saved image: " << sResultFilename << std::endl;
	cudaDeviceReset();
exit(EXIT_SUCCESS);
```

C'est en étudiant la documentation NPP que nous avons trouvé une fonction de filtrage de sobel horizontal :

```
nppiFilterSobelHoriz_8u_C1R(oDeviceSrc.data(),
oDeviceSrc.pitch(),    oDeviceDst.data(),    oDeviceDst.pitch(),
oSizeROI));
```

« `oDeviceSrc` » correspond en fait à l'image chargée du CPU de type « `npp::ImageCPU_8u_C1`» c'est le type NPP pour dire que l'on a une image codé sur 8 bits non signés sur un seul canal (gris). « `oDeviceDst` » est de type « `npp::ImageNPP_8u_C1`» et correspond à l'mage créée par NPP selon l'image source. Enfin « `oSizeROI` » est le masque que l'on va appliquer pour le filtrage défini par la taille « `NppiSize oMaskSize = {3, 3};` ». Enfin « `cudaDeviceReset()` » est une fonction spécifique à CUDA qui indique que la tâche a été effectuée et que l'on a plus besoin de la carte graphique pour le calcul.

Pour ce qui est du filtrage de Sobel à l'aide de OpenCV voici le code :

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
using namespace cv;
using namespace std;
int main(int, char** argv)
{
	Mat src, dst;
	Mat kernel;
	Point anchor;
	double delta;
	int ddepth, kernel_size, dx, dy;
	const char* window_name = "filter2D Demo";
	string Filename = "Lena.pgm";
	src = imread(Filename, CV_LOAD_IMAGE_UNCHANGED);
	if (src.empty())
	{
		return -1;
	}
	namedWindow(window_name, WINDOW_AUTOSIZE);
	delta = 0;
	ddepth = CV_8U;
	dx = 0;
	dy = 1;
	kernel_size = 3;
	clock_t begin = clock();
	int count = 1;
	for (unsigned int i = 0; i < count; i++)
	{
		Sobel(src, dst, ddepth, dx, dy, kernel_size);
	}
	clock_t end = clock();
	float diffms = ((end - begin) * 1000) / CLOCKS_PER_SEC;
	diffms = diffms / count;
	imshow(window_name, dst);
	waitKey(0);
	destroyWindow(window_name);
	string ResultFilename = Filename;
	string::size_type dot = ResultFilename.rfind('.');
	if (dot != string::npos)
	{
		ResultFilename = ResultFilename.substr(0, dot);
	}

	ResultFilename += "_boxFilter.pgm";
```

```
        imwrite(ResultFilename, dst);
        return 0;
}
```

OpenCV utilise des types spécifiques « `Mat` » pour définir une image. Après la création de la fenêtre de visualisation (que l'on n'avait pas dans le filtrage en GPU) on peut définir la variable « `ddepth` » qui indique le format des pixels de l'image. La fonction « `Sobel(src, dst, ddepth, dx, dy, kernel_size)` » demande en plus les degrés de dérivation, comme l'on veut un filtrage horizontal de taille 3 on définit :

```
dx = 0; dy = 1; kernel_size = 3;
```

Nous avons eu des soucis pour l'analyse des temps de calcul, ce qui explique la partie du code où l'on récupère le temps d'exécution à l'aide de la librairie « time.h ».

Figure 14 - Comparaison des images entre GPU (à gauche) et CPU (à droite)

Bien heureusement les images ressorties sont identiques et on observe bien que le filtre fait ressortir les bordures horizontales plutôt que verticales. C'est la comparaison des deux images qui nous permet de valider le fonctionnement du filtre.

Ce qui nous intéresse surtout est la comparaison de performance entre le GPU et le CPU. Pour cela nous nous sommes basés sur l'article de Lee et al. en 2010 (7) qui mesure les performances de nombreuses fonctions courantes en industrie (FFT, convolution, filtre bilatéral, histogramme etc…). Ils proposent une analyse de performance selon de nombreux paramètres : montant mémoire RAM utilisé, cache utilisé, FLOPS (Floating Point operation per Second), consommation énergétique, synchronisation des threads ou encore utilisation de la mémoire partagée.

Nous retiendrons surtout les temps d'exécution des opérations ainsi que la consommation énergétique (en analysant notamment la température). Pour mesure les temps d'exécution nous avons utilisé Visual Studio qui propose des outils d'analyse de performance. Pour ce qui

est de la consommation énergétique, nous avons eu l'idée d'analyser la montée et descente du régime thermique. Ceci permet d'étudier non seulement le pic de température (à peu près équivalent à la consommation) mais aussi le temps que met le composant à stocker et libérer cette énergie (ce qui est très important). Les résultats ont étés rassemblés dans les deux graphique suivants :

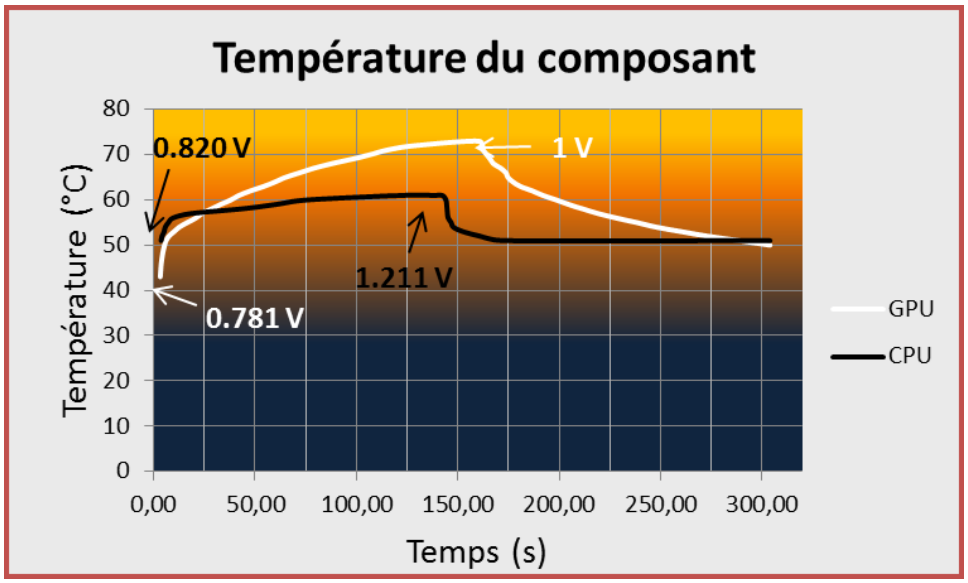


Figure 15 - Régime thermique des composants

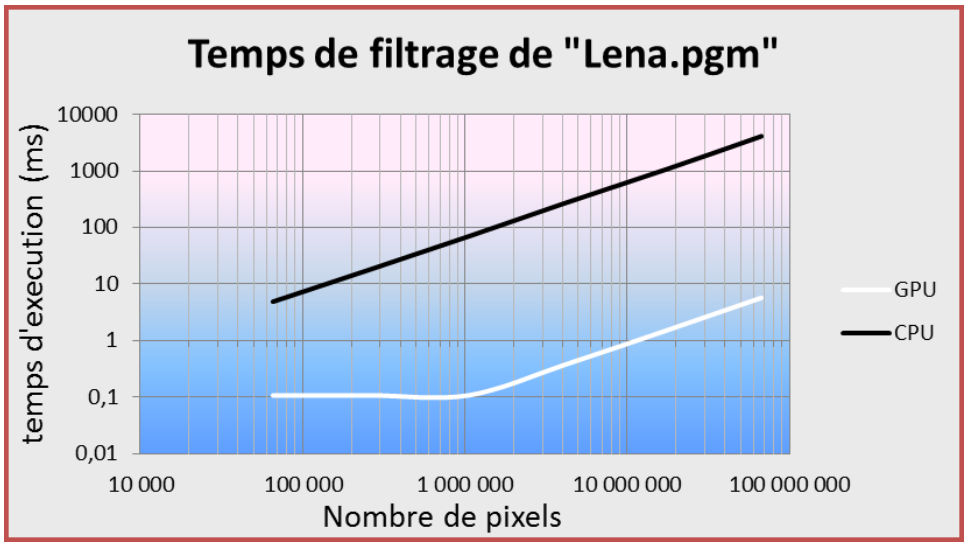


Figure 16 - Temps d'exécution du filtrage de Sobel horizontal

Ces analyses ont été effectuées selon ce que l'on appelle un « stress test », c'est-à-dire que l'on va demander à la ressource une grande quantité de travail d'un seul coup (ce qui se traduit par une boucle de plusieurs milliers de filtrage dans notre cas).

Pour ce qui est de l'analyse du régime thermique, nous avons étudié les températures en lançant le programme et en l'arrêtant au bout de 150s. On remarque que le CPU est beaucoup plus dynamique que le GPU. En effet, il atteint très rapidement son pic

de $60°C$ (au bout de 50s) contrairement au GPU qui lui met 150s pour atteindre $71°C$. La descente du régime pour atteindre la température de repos est semblable à la montée pour les deux systèmes. Ce qu'il est intéressant de remarquer, c'est qu'au repos le GPU consomme moins que le CPU. Ceci est lié au fait que pour un travail « normal » (c'est-à-dire des tâches de bureau sur l'ordinateur type internet, Word etc…) le GPU est très peu sollicité. Cela renforce encore plus l'intérêt du GPGPU qui pourrait alors recevoir plus de tâches du CPU durant des périodes « calmes ». L'analyse de ce régime thermique est semblable à la charge/décharge d'un condensateur en électricité. C'est ce qui explique que de nombreux thermiciens cherchent à calculer la « capacité thermique », qui représente la capacité d'un composant à évacuer la chaleur par le radiateur (dans le cas d'un ordinateur portable standard). Nous avons aussi mesuré la tension en V donne aussi une indication très approximative de la puissance consommée.

Pour les temps d'exécution, nous avons créé plusieurs images « Lena.pgm » avec différentes résolutions :

- $256 \times 256 = 65\,536$ pixels
- $512 \times 512 = 262\,144$ pixels
- $1024 \times 1024 = 1\,048\,576$ pixels
- $2048 \times 2048 = 4\,194\,304$ pixels
- $4096 \times 4096 = 16\,777\,216$ pixels
- $8192 \times 8192 = 67\,108\,864$ pixels

On remarque que le GPU est largement en tête devant le CPU. Celui-ci met déjà 7 ms pour effectuer un filtrage sur une image 256×256 alors que le GPU ne met que 0.1 ms. Les courbes ont tendances à avoir une croissance exponentielle (typique de l'analyse de temps d'exécution en général). Ce qu'il est intéressant de remarquer avec le GPU c'est qu'il met autant de temps pour une image 256×256 que 1024×1024. Ceci est lié à la répartition des tâches dans le GPU : le nombre de cœurs étant élevé, le GPU à la possibilité de faire le filtrage de l'image en un seul temps de cycle de son horloge.

Il y a donc une erreur sur ce graphique dû à cette remarque et il est nécessaire de retrouver la taille maximale de l'image que peut traiter le GPU en un cycle d'horloge. En s'aidant des caractéristiques du constructeur on sait que l'on peut traiter 24.8 milliards de pixel en une seconde. Le temps de calcul de l'image est de 0.1073 ms ce qui équivaut à 2.66 millions de pixels est donc une image 1631×1631 ce qui nous amène au nouveau graphique :

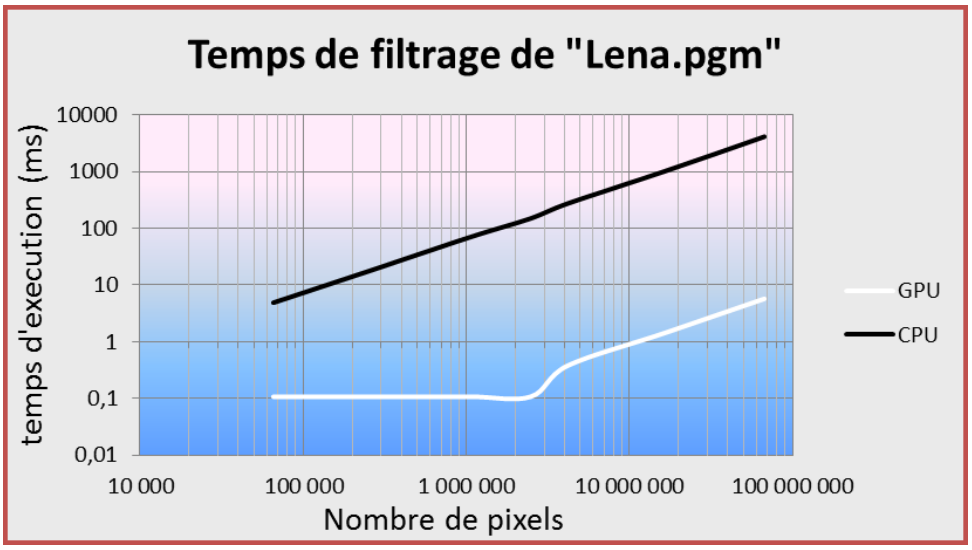


Figure 17 - Nouveau temps d'exécution du filtrage de Sobel horizontal

Conclusion

Les GPUs offrent des capacités d'accélération des applications impressionnantes. Nos expérimentations nous ont permis de comprendre et d'observer les différences de performances entre CPU et CPU. Grâce à la plateforme CUDA développée par NVIDIA, les atouts du GPU computing sont également accessibles à tous. La diversité des méthodes de programmation permet à n'importe quel ingénieur maîtrisant le C/C++ d'exploiter les ressources de cartes graphiques sans formation en graphisme requise. Après bientôt 10 ans de CUDA, l'industrie a déjà adopté le GPU computing et nous comprenons à présent qu'il constitue effectivement l'avenir de l'accélération des applications. Le parallélisme est une nouvelle façon de penser pour les développeurs et une nouvelle base pour construire en entreprise.

La découverte de ce domaine a étendu nos horizons dans le domaine de la vision par ordinateur mais également en informatique de manière générale. Ces connaissances pourront dorénavant être cultivées dans le cadre de projets au Laboratoire de Traitement de l'Information en Santé (LATIS) de l'ETS. Une première application de solveur d'équation linéaire par GPU computing est envisagée, pour ensuite accélérer le traitement d'un recalage échographique pour de la reconstruction 3D.

Recommandations

Pour ceux voulant s'initier au GPGPU, la première question à se poser est de savoir s'il est utile de le faire. En effet, pourquoi se compliquer la tâche si nous avons déjà un programme ne nécessitant pas d'amélioration de performance et fonctionnant très bien actuellement ? Le GPGPU peut être relativement coûteux non seulement en temps mais aussi en argent (compter dépasser $1\,000€$ pour une carte haute gamme).

Ceux qui ont fait une étude et qui en conclut que leur logiciel se doit d'être accéléré doive alors choisir le constructeur : AMD ou NVIDIA. Le premier sera utile dans le cas où les ingénieurs connaissent la programmation par OpenCL, les cartes AMD étant compatibles et optimisés pour OpenCL. NVIDIA propose aussi une compatibilité avec OpenCL mais moins performante que le concurrent. Cette compatibilité OpenCL pour AMD n'est pas à l'avantage du futur acheteur en termes de coût de carte. En effet, les cartes hautes performances de AMD sont beaucoup plus chères que NVIDIA dû à une activité que l'on appelle le « mining[4] ».

NVIDIA à l'avantage d'être de meilleur rapport qualité/prix. Pour ceux qui choisissent NVIDIA, nous ne pouvons que conseiller de commencer par programmer à l'aide de OpenACC. Les résultats vont permettre d'avoir une idée globale du gain de performance que l'on peut obtenir avec un GPU. Ensuite, si le gain de temps est réel, alors une programmation à l'aide des librairies CUDA permettra de très bien accélérer les programmes. Ceux nécessitant de créer des fonctions bien spécifiques à leur travail devront alors se plonger dans la programmation de kernels spécifique CUDA.

Pour vous initier aux méthodes de programmation, jetez un œil aux diverses références qui contiennent notamment des liens vers des tutoriaux. De plus, vous retrouverez de nombreuses vidéos tutoriaux sur internet. Enfin, nous déconseillons aux utilisateurs d'utiliser des librairies GPGPU proposés par des tiers (non officialisés par CUDA), comme par exemple les librairies GPU de OpenCV qui s'avère être de faible performance. Si vous avez vraiment besoin de fonctions qui ne sont pas disponibles dans les librairies proposés dans ce rapport, alors codez-les vous-même !

[4] Le « mining » est une pratique qui a été démocratisé avec l'arrivée des monnaies virtuelles (Bitcoin, Terracoin, Litecoin etc…). Sans rentrer dans les détails, ces monnaies « demande » à des internautes spécifiques (compétents en informatique en général) de vérifier les porte-monnaie cryptés des utilisateurs. Ce sont ces internautes qui vont vérifier que tel utilisateur à bien telle quantité en « prêtant » les ressources de leur PC (et les cartes graphiques !). La plupart des logiciels de « mining » sont codés à l'aide d'OpenCV ce qui explique le coût des cartes graphiques hautes performances AMD.

Références

1. **Barron, E. T., & Glorioso, R. M.** A micro controlled peripheral processor. s.l. : In Conference record of the 6th annual workshop on Microprogramming (pp. 122-128). ACM., 1973.

2. **Altaber, J., Innocenti, P. G., & Rausch, R.** *Multi-microprocessor architecture for the LEP storage ring controls.* s.l. : No. CERN-SPS-85-28-ACC, 1985.

3. **Shen, G., Zhu, L., Li, S., Shum, H. Y., & Zhang, Y. Q.** *Accelerating video decoding using GPU.* s.l. : Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on (Vol. 4, pp. IV-772). IEEE., 2003.

4. **Krüger, J., & Westermann, R.** *Linear algebra operators for GPU implementation of numerical algorithms.* s.l. : ACM Transactions on Graphics (TOG) (Vol. 22, No. 3, pp. 908-916). ACM., 2003.

5. **Buck, I.** *Gpu computing: Programming a massively parallel processor .* s.l. : Code Generation and Optimization. CGO'07. International Symposium on (pp. 17-17). IEEE., 2007.

6. **Ino, F., Matsui, M., Goda, K., & Hagihara, K.** *Performance study of LU decomposition on the programmable GPU.* s.l. : High Performance Computing–HiPC 2005 (pp. 83-94). Springer Berlin Heidelberg, 2005.

7. **Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... & Dubey, P.** *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU.* s.l. : ACM SIGARCH Computer Architecture News (Vol. 38, No. 3, pp. 451-460). ACM, 2010.

8. **Wu, G., Greathouse, J. L., Lyashevsky, A., Jayasena, N., & Chiou, D.** *GPGPU performance and power estimation using machine learning.* s.l. : High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on (pp. 564-576), 2015.

9. **Wu, Q., Ha, Y., Kumar, A., Luo, S., Li, A., & Mohamed, S.** *A heterogeneous platform with GPU and FPGA for power efficient high performance computing.* s.l. : Integrated Circuits (ISIC), 2014 14th International Symposium on (pp. 220-223). IEEE., 2014.

10. **Meihua, L., Yang, H., Koizumi, K., & Kudo, H.** *Fast cone-beam CT reconstruction using CUDA architecture.* 2007.

11. **Moore, N., Leeser, M., & Smith King, L.** *Efficient template matching with variable size templates in CUDA.* s.l. : Application Specific Processors (SASP), 2010 IEEE 8th Symposium on (pp. 77-80). IEEE., 2010.

12. **Grauer-Gray, S., Killian, W., Searles, R., & Cavazos, J.** *Accelerating financial applications on the GPU.* s.l. : Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (pp. 127-136). ACM., 2013.

13. **NVIDIA.** *Whitepaper on NVIDIA's Next Generation CUDA Compute Architecture : Fermi.* 2009.

14. —. Specifications GTX 560M. [En ligne] 30 05 2011. [Citation : 14 04 2015.] http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-560m/specifications.

15. —. Specifications Geforce 840M. [En ligne] 24 08 2014. [Citation : 13 04 2015.] http://www.geforce.com/hardware/notebook-gpus/geforce-840m/specifications .

16. —. About Cuda. [En ligne] 15 07 2014. [Citation : 11 04 2015.] https://developer.nvidia.com/about-cuda.

Table des figures

Annexes

Fichier « Filtrage_Sobel_GPU.cpp »

```
#include <ImagesCPU.h>
#include <ImagesNPP.h>
#include <ImageIO.h>
#include <Exceptions.h>
#include <string.h>
#include <fstream>
#include <iostream>
#include <time.h>
#include <cuda_runtime.h>
#include <npp.h>
#include <helper_string.h>
#include <helper_cuda.h>

inline int cudaDeviceInit(int argc, const char **argv)
{
	int deviceCount;
	checkCudaErrors(cudaGetDeviceCount(&deviceCount));
	if (deviceCount == 0)
	{
		std::cerr << "CUDA error: no devices supporting CUDA." << std::endl;
		exit(EXIT_FAILURE);
	}
	int dev = findCudaDevice(argc, argv);
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, dev);
	std::cerr << "cudaSetDevice GPU" << dev << " = " << deviceProp.name <<
std::endl;
	checkCudaErrors(cudaSetDevice(dev));
	return dev;
}
bool printfNPPinfo(int argc, char *argv[])
{
	const NppLibraryVersion *libVer = nppGetLibVersion();
	printf("NPP Library Version %d.%d.%d\n", libVer->major, libVer->minor, libVer-
>build);
	int driverVersion, runtimeVersion;
	cudaDriverGetVersion(&driverVersion);
	cudaRuntimeGetVersion(&runtimeVersion);
	printf("  CUDA Driver  Version: %d.%d\n", driverVersion / 1000, (driverVersion %
100) / 10);
	printf("  CUDA Runtime Version: %d.%d\n", runtimeVersion / 1000, (runtimeVersion
% 100) / 10);
	bool bVal = checkCudaCapabilities(1, 0);
	return bVal;
}
int main(int argc, char *argv[])
{
	printf("%s Starting...\n\n", argv[0]);
	try
	{
		std::string sFilename;
		char *filePath;
		cudaDeviceInit(argc, (const char **)argv);
		if (printfNPPinfo(argc, argv) == false)
```

```
		{
			cudaDeviceReset();
			exit(EXIT_SUCCESS);
		}
		if (checkCmdLineFlag(argc, (const char **)argv, "input"))
		{
			getCmdLineArgumentString(argc, (const char **)argv, "input",
&filePath);
		}
		else
		{
			filePath = sdkFindFilePath("Lena.pgm", argv[0]);
		}
		if (filePath)
		{
			sFilename = filePath;
		}
		else
		{
			sFilename = "Lena.pgm";
		}
		int file_errors = 0;
		std::ifstream infile(sFilename.data(), std::ifstream::in);

		if (infile.good())
		{
			std::cout << "boxFilterNPP opened: <" << sFilename.data() << ">
successfully!" << std::endl;
			file_errors = 0;
			infile.close();
		}
		else
		{
			std::cout << "boxFilterNPP unable to open: <" << sFilename.data()
<< ">" << std::endl;
			file_errors++;
			infile.close();
		}
		if (file_errors > 0)
		{
			cudaDeviceReset();
			exit(EXIT_FAILURE);
		}
		std::string sResultFilename = sFilename;
		std::string::size_type dot = sResultFilename.rfind('.');
		if (dot != std::string::npos)
		{
			sResultFilename = sResultFilename.substr(0, dot);
		}
		sResultFilename += "_boxFilter.pgm";
		if (checkCmdLineFlag(argc, (const char **)argv, "output"))
		{
			char *outputFilePath;
			getCmdLineArgumentString(argc, (const char **)argv, "output",
&outputFilePath);
			sResultFilename = outputFilePath;
		}
		npp::ImageCPU_8u_C1 oHostSrc;
		npp::loadImage(sFilename, oHostSrc);
		npp::ImageNPP_8u_C1 oDeviceSrc(oHostSrc);
		NppiSize oMaskSize = { 5, 5 };
```

```
            NppiSize oSizeROI = { (int)oDeviceSrc.width() - oMaskSize.width + 1,
(int)oDeviceSrc.height() - oMaskSize.height + 1 };
            npp::ImageNPP_8u_C1 oDeviceDst(oSizeROI.width, oSizeROI.height);
            NppiPoint oAnchor = { 0, 0 };
            clock_t begin = clock();
            int count = 10000;
            for (unsigned int i = 0; i < count; i++)
            {
                  NPP_CHECK_NPP(
                        nppiFilterSobelHoriz_8u_C1R(oDeviceSrc.data(),
oDeviceSrc.pitch(),
                        oDeviceDst.data(), oDeviceDst.pitch(),
                        oSizeROI));
            }
            clock_t end = clock();
            float diffms = ((end - begin) * 1000) / CLOCKS_PER_SEC;
            diffms = diffms / count;
            npp::ImageCPU_8u_C1 oHostDst(oDeviceDst.size());
            oDeviceDst.copyTo(oHostDst.data(), oHostDst.pitch());

            saveImage(sResultFilename, oHostDst);
            std::cout << "Saved image: " << sResultFilename << std::endl;

            cudaDeviceReset();
            exit(EXIT_SUCCESS);
      }
      catch (npp::Exception &rException)
      {
            std::cerr << "Program error! The following exception occurred: \n";
            std::cerr << rException << std::endl;
            std::cerr << "Aborting." << std::endl;
            cudaDeviceReset();
            exit(EXIT_FAILURE);
      }
      catch (...)
      {
            std::cerr << "Program error! An unknow type of exception occurred. \n";
            std::cerr << "Aborting." << std::endl;
            cudaDeviceReset();
            exit(EXIT_FAILURE);
            return -1;
      }

      return 0;
}
```