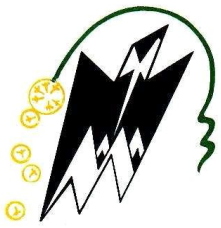
République Algérienne Démocratique et Populaire

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

Université Mouloud Mammeri De Tizi-Ouzou



Faculté de Génie Electrique et d'Informatique

Département de l'Electronique

En vue d'obtention du diplôme Master en Electronique

Option : Réseaux et Télécommunications

THEME :

IMPLEMENTATION DE LA PLATEFORME CUDA POUR LA COMPRESSION JPEG

Travail réalisé par :

M^{me}: DAID née HARIM Khalida

M^{elle}: HADDOUS Karima

Proposé et dirigé par :

Mr: LAHDIR Mourad

Promotion: 2014

Liste de figures :

Liste de tableaux

Sommaire

Chapitre II :Introduction au calcul parallèle

Chapitre III :Installation matérielle et logicielle de la plate forme CUDA

Chapitre IV :Application de CUDA pour la compression JPEG

Introduction

Le calcul haut performance est un domaine qui rassemble les outils matériels et logiciels nécessaires à l'exécution d'applications et de techniques complexes (simulations, calcul numérique, modélisations). Les développements, dans ce domaine, évoluent en parallèle avec les avancées technologiques réalisées dans le domaine des systèmes distribués et architectures parallèles. Les supercalculateurs, les clusters, les grilles de calcul et dernièrement les processeurs graphiques (GPUs) sont des infrastructures de calcul intensif qui subissent des évolutions permanentes permettant de repousser les limites en terme de puissance de calcul. Actuellement la puissance de calcul atteinte par certaines infrastructures parallèles est de l'ordre de Pétaflops (10^{15} opérations flottantes par seconde).

Avec l'ouverture des interfaces de programmation (API) des GPUs, de nombreux chercheurs ont entrepris de les exploiter pour booster les traitements, habituellement destinés aux CPUs. Les calculs peuvent donc être portés sur GPU, mais le mode de fonctionnement SIMD de ces derniers impose la nécessité d'une grande quantité de données en entrée. En effet, ces processeurs tirent leurs forces du recouvrement des threads inactifs (en attente de données à distance du cœur) par l'exécution d'autres threads actifs (dont les donné es sont disponibles). S'il n'y a pas suffisamment de données, les threads seront en nombre insuffisant pour assurer un fonctionnement `a plein régime du processeur graphique, et celui-ci verra ses performances se dégrader.

Les algorithmes de visualisation et de traitement d'images sont à la fois de gros consommateurs de puissance de calcul et de mémoire. Au-dela du rendu 2D/3D, la majorité de ces algorithmes contiennent des phases qui consistent en des calculs similaires entre les pixels de l'image ce qui se prˆété bien `a une parallélisassions sur GPU.

Dans le cadre de notre travail, nous avons choisi de faire une installation d'une plate forme CUDA pour la compression d'image par JPEG.

Ce mémoire est organisé comme suit :

Le premier chapitre de ce mémoire est consacré à la description de l'architecture des GPU. Cependant, afin d'avoir une approche simple de ces architectures et de mieux comprendre leurs évolutions, la première partie fait une rétrospective des microprocesseurs (CPU), de

leurs fonctionnement et des limitations actuelles notamment dues aux limites technologiques et physiques.

Dans le chapitre deux, on trouve la définition de CUDA avec sa taxonomie ; l'organisation des données dans les GPUS et l'évolution de ces derniers avec les différents domaines d'applications.

Le chapitre trois, donne les différentes étapes d'installation de notre plate forme et la mise en marche de CUDA avec plusieurs exemples livrés par CUDA SAMPLES.

Le dernier chapitre, nous présentons les tests et les résultats pour la compression d'images par JPEG.

Enfin, on termine notre mémoire par une conclusion générale et des perspectives.

Chapitre I

I.1-Préambule :

Depuis l'apparition de micro processeur, un rythme d'évolution très rapide s'est installé. Les avancées successives de l'électronique et des techniques de fabrications, concernant notamment les progrès dans le domaine de la conduction des matériaux semi-conducteurs, vont permettre d'améliorer la fiabilité, les performances et la miniaturisation des processeurs.

Ce rythme d'évolution s'est maintenu jusqu'à aujourd'hui faisant apparaître de nombreux modèles de plus en plus puissants et de plus en plus complexes.

Une "loi" tenant compte de l'évolution technologique des circuits intégrés et de l'évolution de l'architecture des matériels informatiques à été énoncée par G.Moore en 1965 [1] et est considérée comme étant encore valable actuellement. Cette loi prédit que le nombre de transistors sur une puce de silicium double tous les deux ans et à coût constant.

La Fig. 1.1 montre l'évolution du nombre de transistors gravés sur les microprocesseurs entre 1970 et 2010.

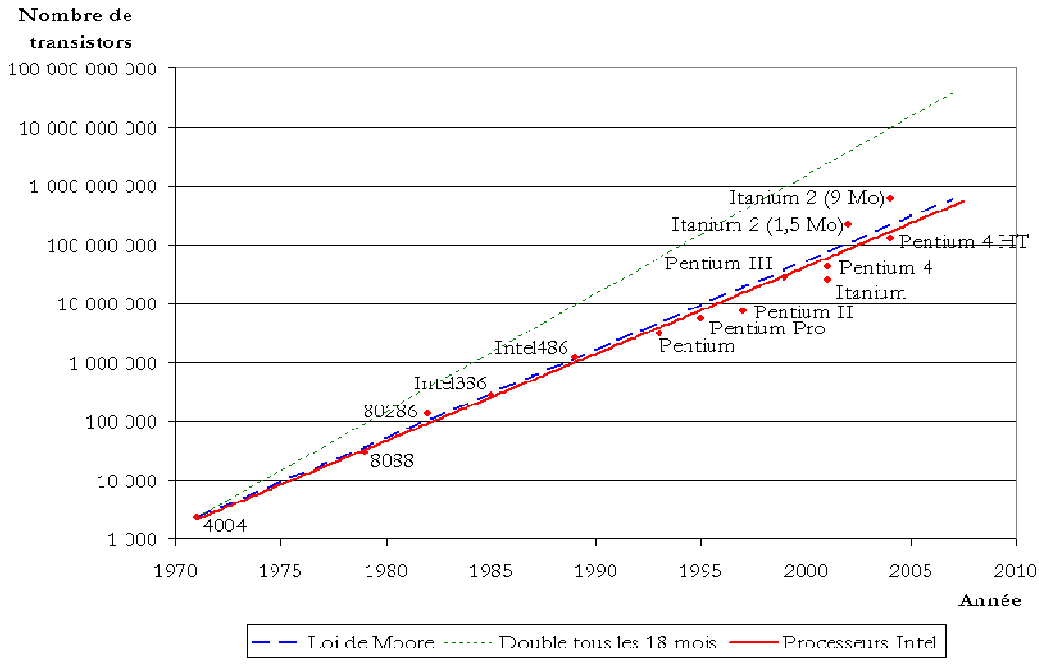


Figure I.1. Evolution du nombre de transistors gravés sur les microprocesseurs entre 1970 et 2010. [1]

I.2- Architecture et fonctionnement du CPU :

Le microprocesseur, aussi appelé CPU (Central Processing Units) est l'élément principal d'un ordinateur. Le microprocesseur doit être en mesure de lire les instructions en mémoire, de les décoder, puis de les exécuter. Ces instructions "machine" sont traitées les unes après les autres et cette suite d'instructions est appelée un programme. Chaque modèle de microprocesseur lit des instructions spécifiques à sa conception sous forme d'un langage de base que l'on appelle assembleur. Ce langage de programmation est complexe est spécifique à la machine et est codé en hexadécimal.

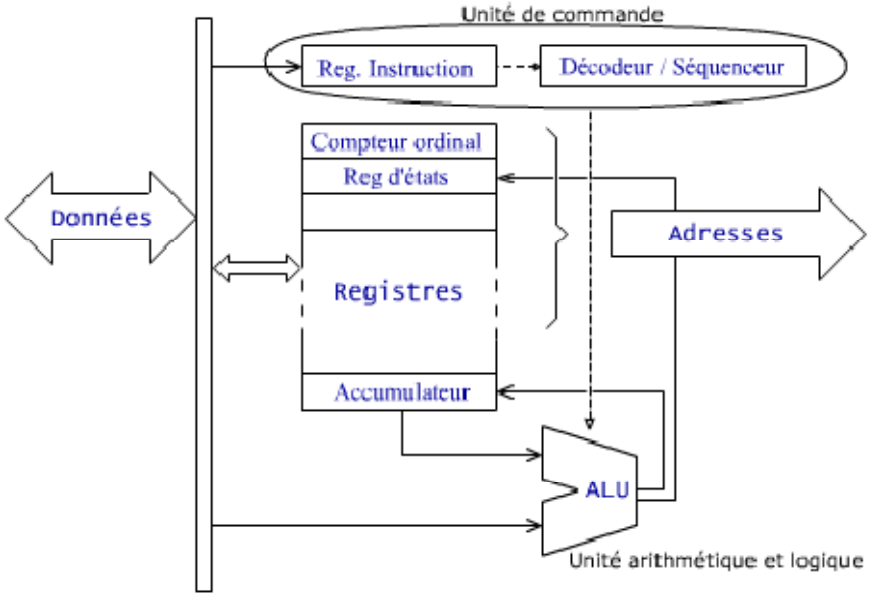


Figure I .2 Architecture du CPU

I.3-Les processeurs mono-cœur :

Un cœur physique est un ensemble de circuits capable d'exécuter des programmes de façon autonome. Toutes les fonctionnalités nécessaires à l'exécution d'un programme sont présentes dans ce cœur : compteur ordinal, registres, unités de calcul, etc.

Le CPU (mono cœur) est constitué essentiellement de trois parties :

- L'unité de commande : Qui cherche les instructions en mémoire, les décode et coordonne le reste du processeur pour les exécuter. Une unité de commande élémentaire se compose essentiellement d'un registre d'instruction et d'une unité «décodeur / séquenceur»

- L'unité Arithmétique et Logique (ALU): Exécute les instructions arithmétiques et logiques demandées par l'unité de commande. Les instructions peuvent porter sur un ou plusieurs opérandes. La vitesse d'exécution est optimale quand les opérandes se situent dans les registres plutôt que dans la mémoire externe au processeur.
- Les registres sont des cellules mémoire interne au CPU : Ils sont peu nombreux mais d'accès très rapide. Ils servent à stocker des variables, les résultats intermédiaires d'opérations (arithmétiques ou logiques) ou encore des informations de contrôle du processeur.

La structure des registres varie d'un processeur à l'autre. C'est ce qui fait que chaque type de CPU a un jeu d'instruction qui lui est propre. Leurs fonctions de base sont néanmoins semblables et tous les processeurs possèdent en gros les mêmes catégories de registres.

- **L'accumulateur** est principalement destiné à contenir les données qui doivent être traitées par l'ALU.
- **Les registres généraux** servent au stockage de résultats intermédiaires
- **Les registres d'adresses** servent à confectionner des adresses de données particulières, ce sont, par exemples, les registres de base et d'index qui permettent entre autre d'organiser les données en mémoire comme des tables indicées.
- **Le registre d'instruction** contient le code de l'instruction qui est traitée par le décodeur /séquenceur.

- **Le compteur ordinal ou *program counter*** contient l'adresse de la prochaine instruction à exécuter. En principe, ce registre ne cesse de compter. Il génère les adresses des instructions à exécuter les unes à la suite des autres. Certaines instructions demandent quelquefois de changer le contenu du compteur ordinal pour faire une "rupture de séquence" c'est à dire un saut ailleurs dans le programme.
- **Le registre d'état** appelé parfois **registre de condition** contient des indicateurs appelés flags (drapeaux) et dont les valeurs (0 ou 1) varient en fonction des résultats des opérations arithmétiques et logiques. Ces états sont utilisés par les instructions des sauts conditionnels.
- **Le pointeur de pile** ou ***stack pointer*** gère certaines données en mémoire en les organisant sous forme de piles.

I.3.1 **-Pipelines d'instructions et architecture super scalaire :**

L'unité de commande élémentaire qui, dans les premiers microprocesseurs, contenait un registre d'instruction et un décodeur / séquenceur, est remplacé depuis les processeurs de la troisième génération (386) par un ou des pipelines. Dans le 486, le pipeline est unique et se compose de cinq unités fonctionnelles.

Elles "travaillent à la chaîne" pour effectuer simultanément les étapes successives de plusieurs instructions.

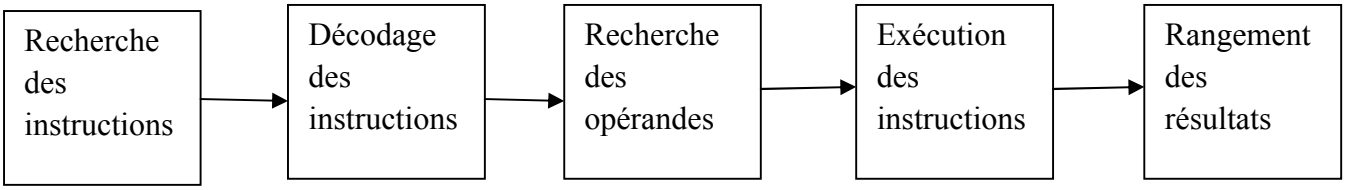


Figure I.3 - Pipelines d'instructions et architecture superscalaire

La première unité fonctionnelle du pipeline va chercher les instructions en mémoire et les range dans un tampon, la seconde unité décode l'instruction et ainsi de suite.

Globalement, il faut toujours autant de temps pour exécuter une instruction mais la vitesse d'exécution est multipliée par cinq puisque avec ce système les instructions suivantes sont entamées dès que possible, les instructions se font donc en parallèle.

Les premiers Pentium (cinquième génération) ont deux pipelines en parallèle ce qui leur permet d'accomplir deux instructions par cycles.

Dans les Pentium actuels, l'architecture «super scalaire» multiplie les chemins des instructions au niveau de certains étages du pipeline, les unités fonctionnelles d'exécutions ont alors recours à plusieurs Unité Arithmétiques et Logiques (ALU).

Le processeur est dit superscalaire de rang "n" s'il possède n unités arithmétiques et logiques.

I.3.2-Caches et accès mémoire :

Si on compare l'augmentation des fréquences des processeurs et celle des mémoires, on se rend compte que, la fréquence de la mémoire augmente bien plus lentement que celle des processeurs. Cela a pour conséquence directe une augmentation des temps d'accès à la mémoire vus par le processeur. Ce phénomène est appelé "le Mur de la mémoire" (ou Memory Wall) selon Wulf *et AL.* . Il faut actuellement de l'ordre de 300 cycles processeur pour accéder à un mot mémoire. Pour masquer de telles latences, les fabricants de processeurs ont conçu des modifications architecturales, et notamment les caches.

Un cache est une mémoire rapide embarquée dans le processeur. Les processeurs utilisent souvent plusieurs niveaux de caches : les plus proches étant plus rapides mais aussi plus petits que ceux plus éloignés du processeur.

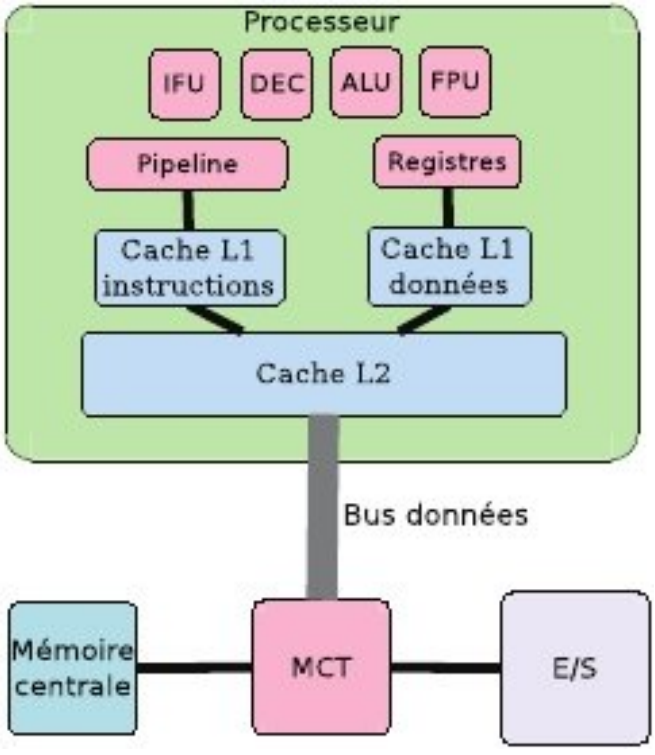


Figure I.4 – Vue d'ensemble simplifiée d'une architecture mono-cœur (typique d'un Intel Pentium mono-cœur), avec ses unités de calculs et ses 2 niveaux de caches.

L'accès à la mémoire centrale et les E/S se font de façon centralisée par le biais du contrôleur mémoire (MCT).

I.3.3-Thread : Séquence (flot) d'exécution du code d'un programme, au sein d'un processus.
Un thread est caractérisé par :

- Sa propre pile d'exécution.
- Identificateur de thread.
- Pointeur d'instruction.

I.3.4- Multi-threading: Plusieurs flots d'exécution simultanés, les différents threads partagent les ressources systèmes.

I.4- Processeurs multi-cœurs :

La chaleur, la dissipation d'énergie et la complexité des processeurs modernes sont des facteurs qui limitent le fonctionnement du mono cœur. C'est pourquoi la dernière décennie a vu l'arrivée des processeurs multi-cœurs sur le marché grand public.

I.4.1Caches privés et partagés :

Cette organisation en cœurs et processeurs a un fort impact sur la hiérarchie mémoire, notamment au niveau des caches. En effet, on voit clairement apparaître ici la notion de caches partagés entre plusieurs cœurs. Jusqu'ici, un cache était privé à un processeur.
L'impact de cette notion de partage de caches a des répercutions importantes sur les performances. Lors d'un accès mémoire, un cœur doit d'abord aller chercher la donnée dans ses propres caches privés. Si elle n'y est pas, au lieu de la chercher directement dans les caches de niveaux supérieurs, il doit avant savoir si elle n'est pas présente dans un des caches privés des autres cœurs. Cela va donc introduire des traitements supplémentaires lors d'une faute de cache.

1.5 -Le calcul parallèle :

Avant de donner le principe d'un calcul parallèle, nous avons juge utile de définir, tout d'abord, celui de son oppose, a savoir le calcul séquentiel et ce, pour mieux cerner la

différence entre eux. Un calcul séquentiel consiste a exécuter un programme, instruction par instruction, par un seul processeur (unité de calcul) et de façon à ce qu'une seule instruction soit exécutée a la fois.

En revanche, un calcul parallèle est définit comme l'exécution d'un ou plusieurs programmes, simultanément, par plusieurs processeurs. Nous avons, en général, deux manières de réaliser un calcul parallèle. La première consiste à découper le programme en plusieurs taches de calcul puis, exécuter toutes ces taches en parallèles par différents processeurs. La seconde nécessite le partitionnement des données du problème à traiter, de manière a ce que chaque partie de données soit attribuée à un processeur différent. Ensuite, tous les processeurs exécutent en parallèle les instructions du même programme mais en opérant sur des données différentes. Cette dernière méthode, appelée la parallélisassions de données.

Le calcul parallèle a pour objectif d'exploiter la grande quantité de ressources (processeurs, mémoires, espaces de stockage, etc) qu'offrent les calculateurs parallèles ; ceci, dans le but de réduire le temps d'exécution des applications nécessitant un long traitement et/ou pour pouvoir exécuter celles portant sur des volumes de données très importants. Tout cela nous permet d'aborder de nouveaux problèmes, de plus en plus, complexes et de tailles toujours croissantes.

I.5.1-Eléments de parallélisme :

a. La taxonomie de Flynn :

Il existe dans la littérature plusieurs classifications portant sur les architectures de calculateurs parallèles et basées sur différents critères de classification. Dans cette section nous présentons la classification la plus largement utilisée dans le domaine du calcul parallèle, a s'savoir : la taxonomie de Flynn [1].

Elle est basée sur deux critères :le nombre d'instructions et le nombre de données qui peuvent être traitées, simultanément, par les différents processeurs du calculateur parallèles.

Les quatre catégories possibles de la taxonomie de Flynn sont décrites ci-après :

- **Instruction unique, donnée unique (SISD)**

La classe SISD (Single Instruction, Single Data) représente l'ensemble des calculateurs séquentiels a une seule unité de calcul (ou monoprocesseur). Ce sont les calculateurs qui ne sont capables de traiter qu'une seule instruction sur une seule donnée, par cycle d'horloge. Bien évidemment, cette catégorie n'est pas une architecture parallèle.

- **Instructions multiples, donnée unique (MISD)**

La classe MISD (Multiple Instruction, Single Data) correspond aux calculateurs parallèles pouvant exécuter plusieurs instructions, simultanément, sur la même donnée.
Peu de calculateurs MISD ont existe en pratique, vu le nombre réduit des applications qui peuvent être mises en œuvre sur ce type d'architecture.

- **Instructions unique, donnée multiples (SIMD)**

La classe SIMD (Single Instruction, Multiple Data) correspond aux processeurs vectoriels et, plus généralement, aux calculateurs composes d'un grand nombre d'unités de calcul. A chaque cycle d'horloge, tous les processeurs d'un calculateur SIM D exécutent simultanément, la même instruction mais opérant sur des données différentes. Cette architecture parallèle est bien adaptée aux traitements des problèmes à structure régulière
ou la même instruction est appliquée a un ensemble de données (exécution des opérations sur des vecteurs ou des tableaux).

- **Instructions multiples, donnée multiples (MIMD)**

La classe MIMD (Multiple Instruction, Multiple Data) représente la catégorie la plus générale dans cette taxonomie. Les calculateurs parallèles MIMD possèdent plusieurs processeurs interconnectes entre eux, tels que chaque processeur soit capable de suivre son propre chemin d'exécution. En effet, a chaque cycle d'horloge, les processeurs différents.

b-Mémoires des architecteurs parallèles :

Nous pouvons distinguer, en général, deux modèles de gestion de la mémoire des calculateurs parallèles : la mémoire partagée et la mémoire distribuée. Ces deux modèles de mémoire permettent de définir les modalités d'accès aux données des autres processeurs dans un calcul parallèle.

- **Mémoire partagée(SHARED MEMORY)**

Dans ce type d'architecture, les processeurs du calculateur parallèle ont un accès direct au même espace mémoire physique via des liens de communication performants, avec un temps d'accès rapide et équitable. En effet, les processeurs peuvent opérer Indépendamment mais toutes les données du calcul parallèle sont placées dans une mémoire commune et ce, de façon `a ce que les changements établis dans la mémoire par un processeur soient immédiatement visibles par les autres processeurs. Dans ce cas, les échanges de données entre processeurs sont effectués via la synchronisation des lectures/ écritures dans la mémoire partagée.

La figure I.5 montre un exemple d'architecture parallèle à mémoire partagée.

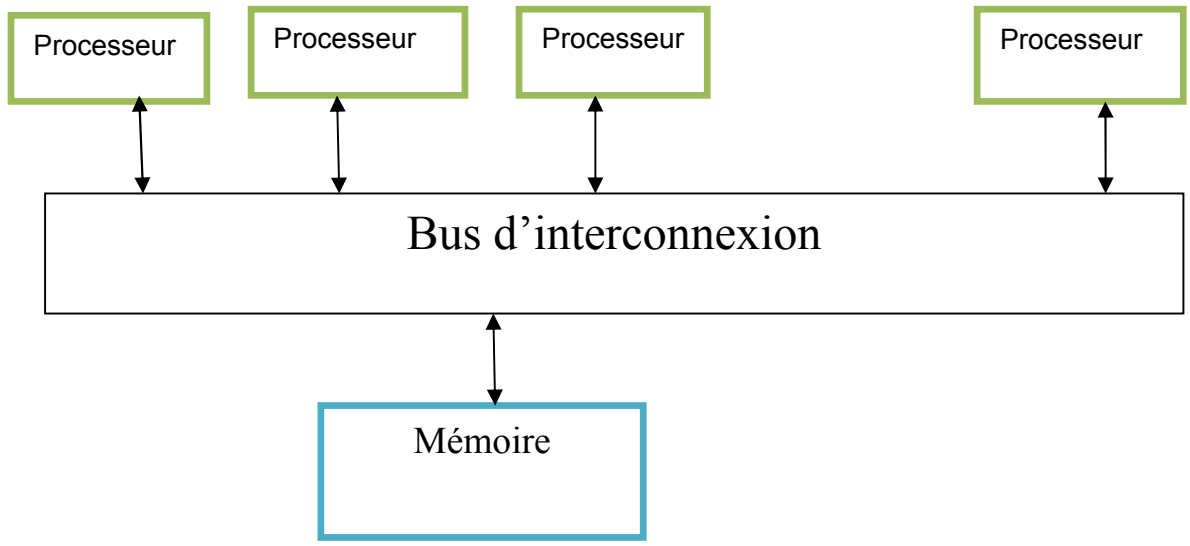


FIGURE I.5 – Exemple d'architecture parallèle à mémoire partagée

- **Mémoire distribuée**

Nous pouvons trouver ce type de mémoire, plus particulièrement, sur les plateformes de calcul parallèle à ressources distribuées, par exemple les grappes et les grilles de calcul. Dans ce cas, chaque processeur de la plateforme parallèle possède sa propre mémoire locale dans laquelle les changements ne sont pas visibles depuis les autres processeurs. Par

conséquent, l'accès aux données des mémoires distantes (mémoires des processeurs voisins) est assuré par des envois de messages entre processeurs via un réseau de communication. La figure I.6 illustre un exemple d'architecture parallèle à mémoire distribuée.

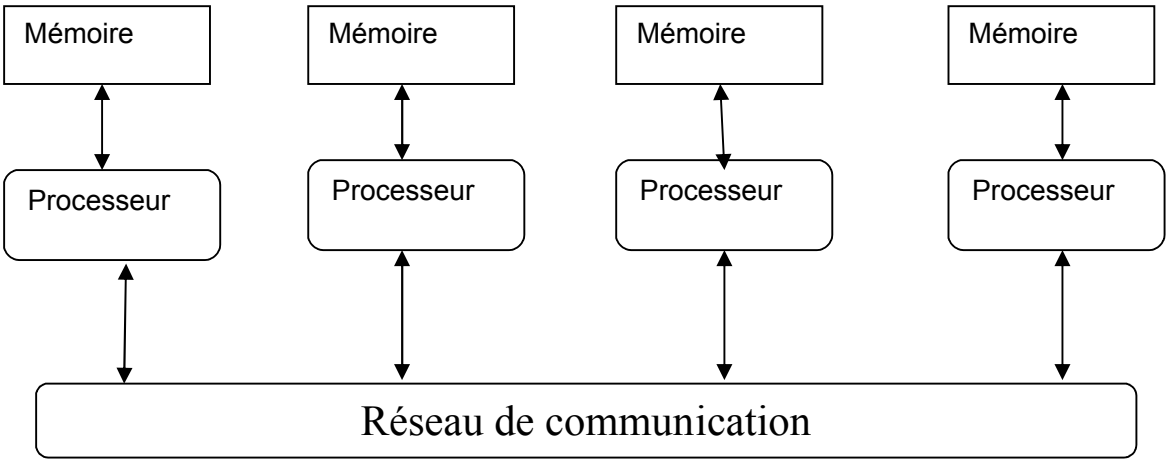


FIGURE I.6 – Exemple d'architecture parallèle à mémoire distribuée

1.6- Les GPU (Graphical Processing Unit) :

Depuis quelques années, les CPU commencent à montrer leurs limites technologiques en terme d'architecture et de vitesse. Leurs architectures multi-cœurs (plusieurs cœurs de calcul au sein d'un même microprocesseur), leur permet de fournir une puissance de calcul toujours plus élevée. Mais cette architecture a une limite qui est liée au temps de latence relativement long lors du transfert des informations entre la mémoire et le microprocesseur.

Autrement dit, la bande passante ou la quantité d'informations transférées par seconde, n'est pas suffisante et est un facteur très limitant pour les performances des CPU. Contrairement aux processeurs graphiques (GPU : Graphic Processing Unit) actuels offrent une très grande puissance de calcul, à un coût relativement réduit.

En effet, l'évolution de ces processeurs est telle que leur puissance dépasse largement celle des CPU. De plus ils sont associés à des mémoires très rapides, offrant une grande bande passante. Depuis quelques années, il est possible de programmer les unités de calcul des GPU et de détourner leur finalité première pour faire du calcul numérique. La puissance brute de calcul proposée par les GPU a largement dépassée depuis quelques années celle affichée par les CPU les plus performants comme le montre la Figure I.8. C'est à partir de 2003 que l'on peut voir la progression (en terme de GB/s) des cartes graphiques (ici, les cartes NVIDIA) par

rapport à l'évolution des CPU. Aujourd'hui, les architectures des cartes graphiques des gammes GeForceGTX montrent une bande passante pouvant être jusqu'à quatre fois supérieures par rapport aux architectures des microprocesseurs les plus récents.

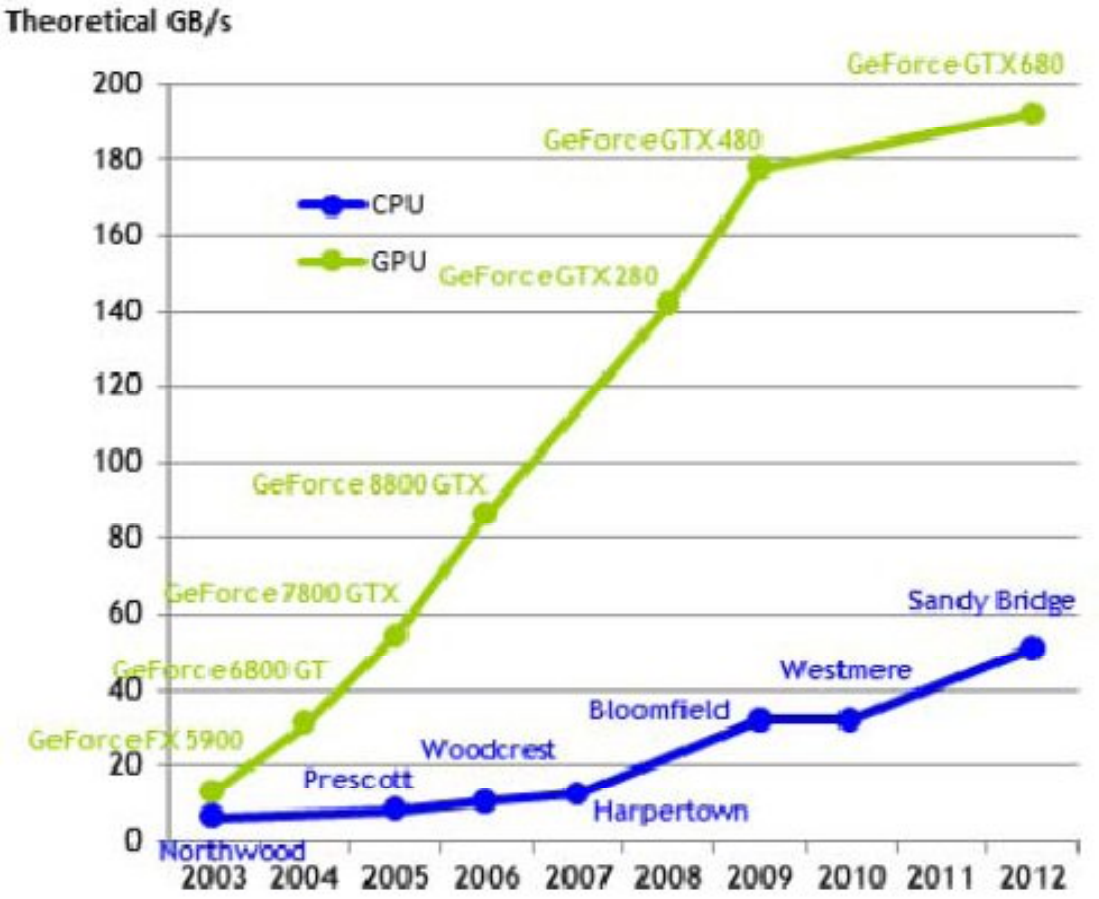


Figure I.8 Comparaison de la bande passante mémoire (en GB/s) pour le CPU (INTEL) et le GPU (NVIDIA) entre 2003 et 2012.

A partir de 2003 les GPU commencent à se démarquer et vont prendre un net ascendant sur les CPU dans les années suivantes (R_ef. [1 ?).

I.6.1-Les différentes générations des GPU :

Les architectures des GPU ont évolué depuis une décennie, franchissant parfois des sauts technologiques importants. Nous nous proposons ici de classer ces architectures en différentes générations.

- **Première génération :** La première génération de GPU à proprement parler a débuté avec l'arrivée des Voodoo Graphics de *3*dfx Interactive en 1996, et dura jusqu'en 1999.

- **Deuxième génération :**Les premières GeForce 256 de NVidia (NV10) font leur apparition en 1999, à peu près en même temps que les Radeon 7500 d'ATI (architecture RV200) et Savage 3D de S3.

- **Troisième génération:** Dès cette génération, les constructeurs NVidia et ATI se partagent la quasi-totalité du marché, NVidia ayant fait l'acquisition de 3dfx. Les Geforce 3 (NV20) en 2001 et GeForce 4 Ti (NV25) en 2002 de NVidia, la Radeon 8500 d'ATI (R200) en 2001 forment cette génération de GPU, permettant enfin au développeur de diriger la transformation des vertex par une suite d'instructions qu'il spécifie.

- **Quatrième génération** Courant 2002, ATI propose sa Radeon 9700 (R300), et NVidia sa GeForce FX (NV30) à la fin de la même année. Ces deux cartes, en plus de supporter le jeu d'instructions DirectX 9 et de nouvelles extensions OpenGL, apportent de plus la possibilité de programmer le traitement des pixels.

- **Cinquième génération** Les GeForce 6 (NV40, avril 2004) et GeForce 7 (G70, juin 2005) de NVidia, les Radeon X800 (R420, juin 2004) et X1800 (R520, octobre 205) et dérivées composent cette génération. Ces cartes permettent quelques fonctions intéressantes

- **Sixième génération :**C'est la génération en plein essor actuellement, équipant la plupart des ordinateurs sur le marché. Ses limites sont plus floues, chaque constructeur apportant ses innovations propres. Elle est composée des GeForce 8 (G80) et GeForce 9 (G92), lancées respectivement en 2006 et 2008, apportant des modifications dans la façon de concevoir le pipeline graphique.

- **Septième génération :** La dernière génération en date n'est pas encore répandue à l'heure actuelle. Les GeForce 200 (G200), lancées en juin 2008, apportent des améliorations principalement techniques : augmentation de la mémoire disponible, du nombre de processeurs, des fréquences mémoire et GPU, de la bande passante par

élargissement du bus de données, pour des performances annoncées jusqu'à deux fois supérieures aux séries précédentes. Pour ATI, les Radeon HD4000, en juin 2008 également, sont censées rivaliser avec la série GeForce 9, mais les performances des modèles haut de gamme les mettent au même niveau que les GeForce 200, proposant les mêmes types d'améliorations techniques, mais affichant des tarifs plus attractifs et une consommation électrique revue à la baisse a compléter.

I.6.2-Architecture classique :

L'architecture des GPU est très spécialisée et la plupart des opérations sont figées. Les données sont traitées massivement en parallèle en appliquant la même opération, dans une unité dédiée, à toutes les données (modèle ou/noyau ou stream processing). C'est cette spécificité qui permet au GPU d'avoir une puissance de calcul très élevée. L'architecture d'un GPU se compose traditionnellement d'un pipeline de calcul (Figure I.9).

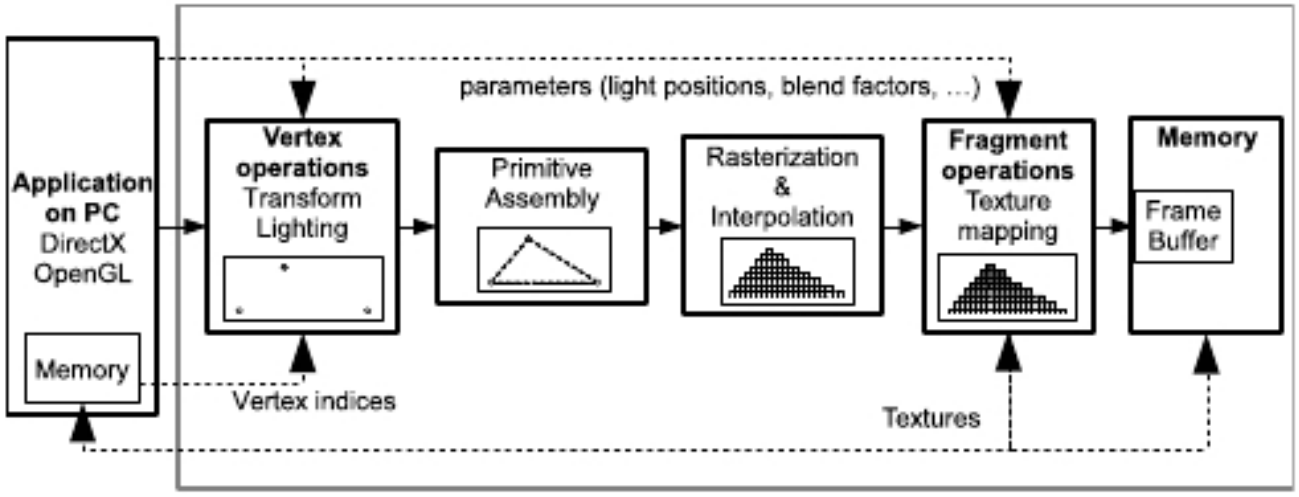


Figure . I.9– Architecture classique d'un GPU

Le processeur de sommets (Vertex processor) calcule des transformations sur des points (position 3D, éclairage, ...) qui sont ensuite assemblées sous forme de triangles, puis transformées en objets affichables (Rasterization). Le processeur de fragments applique la texture sur les objets et calcule le rendu final. La puissance de calcul vient de la spécificité des traitements effectués. Les opérations de contrôle sont très renduites et chaque type d'opération est réalisé dans une unité dédiée.

Dans ce modèle d'architecture, le processeur de sommets et le processeur de fragments sont programmables. Initialement introduits pour créer des effets de rendu personnalisés, les

possibilités de programmation entendent l'utilisation des processeurs graphiques à d'autres types de calcul. Les programmes, appelés shaders, sont donc appliqués sur les sommets et les pixels.

I.6.3-Les différentes architectures des GPU NVIDIA :

La figure ci-dessous montre l'évolution des architectures des GPU NVIDIA .

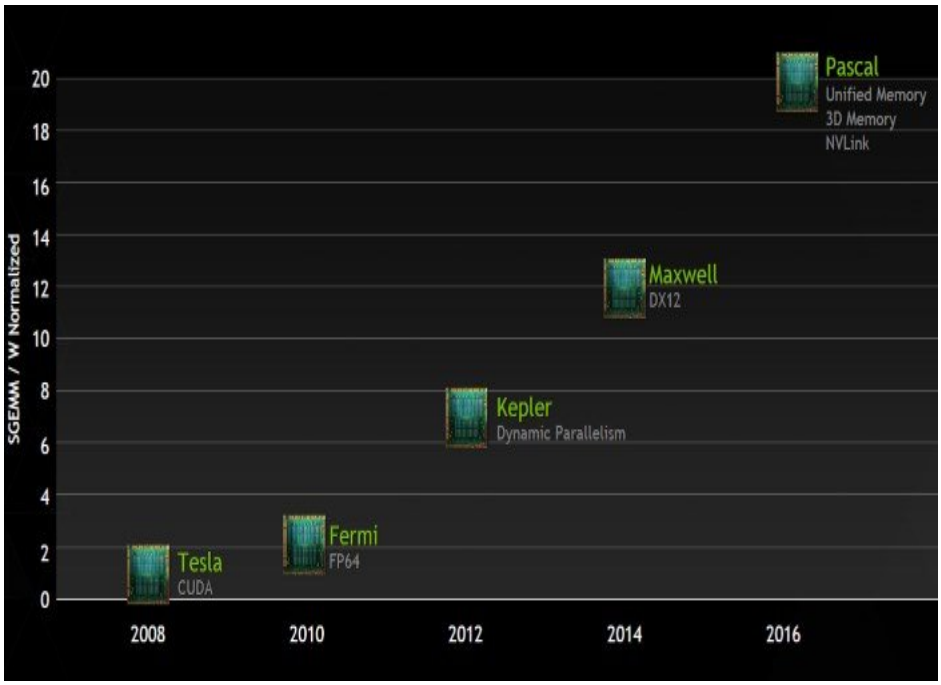


Figure I .10 Les différentes architectures des GPU NVIDIA

- **Architecture Tesla :**

Tesla est une gamme de cartes accélératrices utilisant des processeurs graphiques faisant office de GPGPU produits par NVIDIA et dont le but est d'assister le processeur

central pour les calculs grâce à la bibliothèque logicielle Compute Unified Device Architecture (CUDA).

Les principales particularités des Teslas par rapport aux GeForces dont elles sont issues sont l'absence de sorties vidéos, et l'absence de ventilateurs sur la carte, le flux d'air étant assuré par la ventilation du serveur

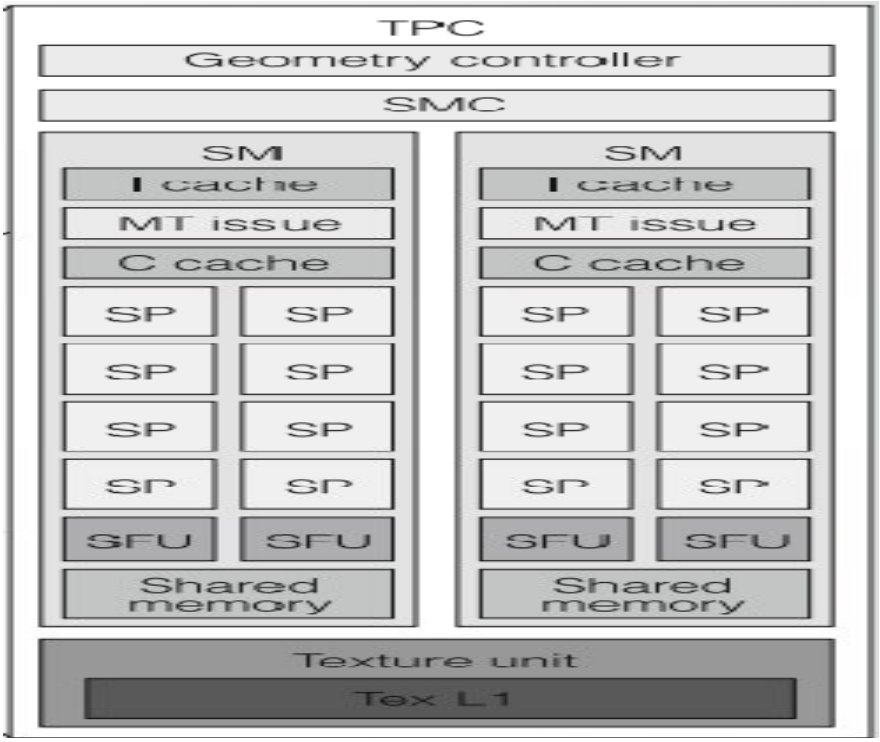
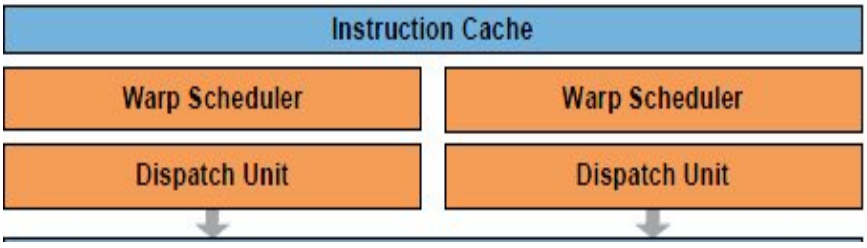


Figure I.11 Architecture Tesla

- **Architecture Fermi :**

En Mars 2010, NVidia sort la GTX 480, la première carte graphique à utiliser l'architecture Fermi avec le GF 100.



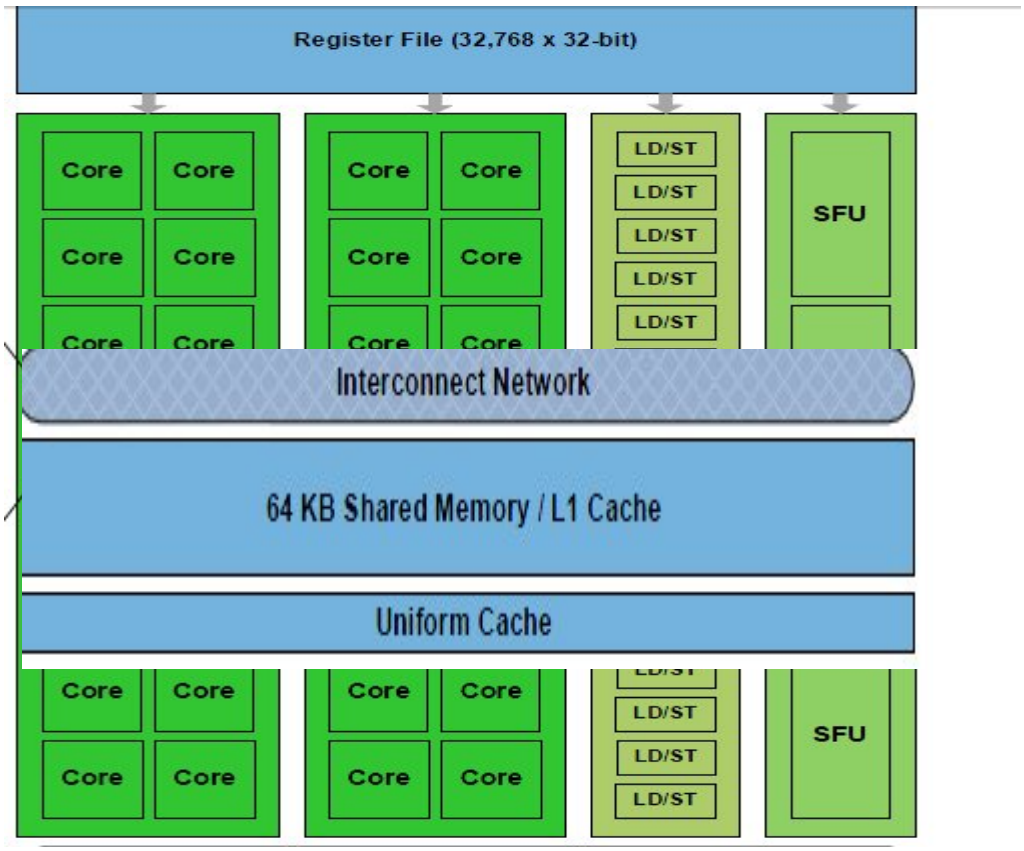


Figure I. 12 Architecture NVIDIA Fermi

- **Architecture Kepler :**

Derrière cette technologie, se cache une couche logicielle destinée au stream processing paradigme de programmation lié au mode SIMD (Single Instruction Multiple Data) permettant à certaines applications d'exploiter la programmation parallèle. Ainsi, dan cette architecture, un GPU est composé de :

- 15 SM
- 1 SM est composé de 192 SP = cœur
- 1 SM possède une mémoire partagée entre tous les SPs.

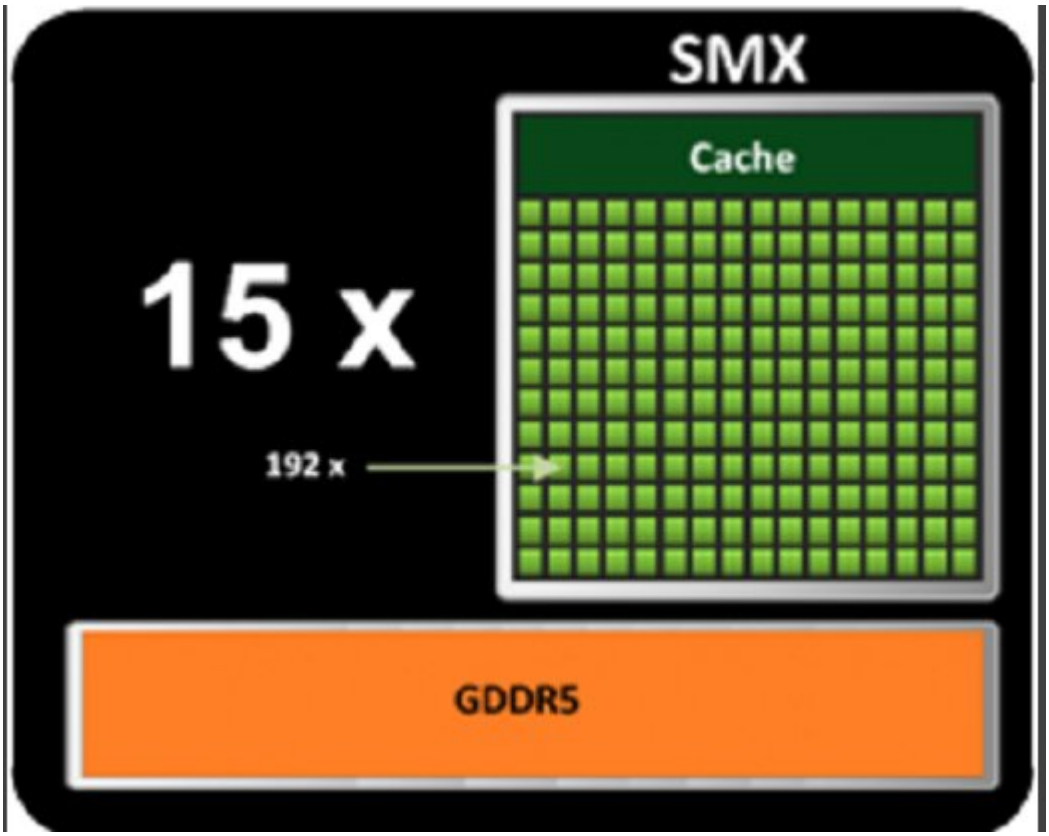


Figure I .13 Architecture de kepler

- **Architecture de Maxwell :**

Comme Kepler, Maxwell est une architecture massivement parallèle constitué de centaines de cœurs CUDA (512 à la GeForce GTX 750, la GeForce 640 en 750 GTX 750 Ti). Maxwell est basé sur le même procédé de fabrication 28nm comme Kepler, mais Maxwell est beaucoup plus économe en énergie.

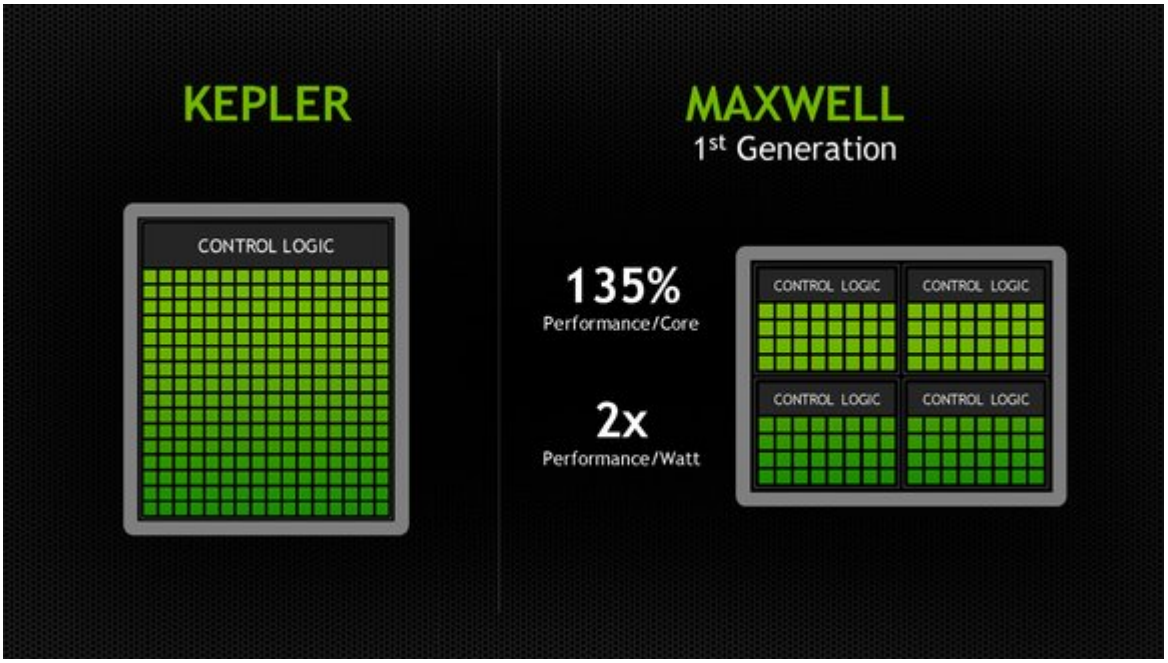


Figure I.14 Architecture de Maxwell

Dans l'architecture Kepler, un logique de contrôle gère tous les cœurs CUDA du GPU. Maxwell a plusieurs éléments logiques de commande, dont chacun gère une poignée de cœurs CUDA.

- **Architecture Pascal :**

  Nvidia vient de dévoiler une nouvelle feuille de route pour ses GPU.

  Après Maxwell cette année, il faudra attendre 2016 et Pascal pour voir arriver une nouvelle architecture GPU.

  Lors de sa précédente conférence GTC, en 2013, Nvidia avait publié une feuille de route faisant état de l'arrivée de l'architecture GPU Maxwell en 2014 puis de Volta à une date non indiquée.

  Maxwell est bel et bien arrivé cette année avec la GeForce GTX 750 Ti, mais il faudra être plus patient pour voir arriver Volta.

I.7-Discussion : Dans ce chapitre nous avons vu, les différentes architectures des CPU_S et GPU_S avec leurs différentes générations. Dans un premier temps, nous avons définit le principe général d'un calcul parallèle et donner quelques caractéristiques des architectures parallèles à mémoire partagée et celles à mémoire distribuée permettant d'effectuer ce type de calcul. Pour cela, nous avons présenté les quatre classes d'architectures informatiques proposées par Flynn.

II -1Préambule :

En 2006, NVIDIA a proposé CUDA (Compute Unified Device Architecture), un environnement de programmation parallèle qui s'apparente à un logiciel permettant de manipuler de façon transparente la puissance d'une carte graphique composée de plusieurs multiprocesseurs, tout en étant facile de prise en main pour des utilisateurs familiers avec le langage C. Les langages C, C++ et Fortran peuvent être utilisés conjointement dans les fonctions (qui sont encore appelées kernels), voir la Figure II.1 La majorité des cartes graphiques grand public NVIDIA sont compatibles CUDA. CUDA s'est imposé comme un des principaux langages standards de manipulation des GPUs.

II.2-Définition :

a- un thread **:** Un thread consiste en un élément de base des données à traiter ; les threads CUDA sont extrêmement « légers » par rapport aux threads CPU.

b- b-Warp **:** Un warp en CUDA est un ensemble de 32 threads, il s'agit de la taille minimale .des données traitées de façon SIMD par un multiprocesseur en CUDA.

c- Blocs : composé de plusieurs threads exécutés de manière concurrente.

d- Une grille : composée de plusieurs blocs de threads exécutés de manière concurrente.

e-Host ou CPU : c'est l'ordinateur qui sert d'interface avec l'utilisateur et qui contrôle le « device » utilisé pour exécuter les parties de calcul intensif basé sur un parallélisme de données. L'hôte est responsable de l'exécution des parties séquentielles de l'application.

f-Device est le GPU connecté à « l'hôte » et qui va exécuter les parties de calcul intensif basé sur un parallélisme de données.Le device, ou périphérique, est responsable de l'exécution de la partie parallèle de l'application **kernel** est une fonction qui peut être appelée depuis
« l'hôte » et qui est exécutée en parallèle sur le « device » CUDA par de nombreuses threads.

Le « kernel » est exécuté simultanément par des milliers de threads

II.3-Répartition des threads :

Un noyau est exécuté comme une grille de blocs de threads.

Tous les threads partagent l'espace donnée mémoire.

Un bloc de threads est un ensemble de threads qui peuvent coopérer ensemble en synchronisant leur exécution pour des accès sans aléas à la mémoire partagée

Partager efficacement les données via une mémoire partagée à faible latence

Deux threads de deux blocs différents ne peuvent coopérer .

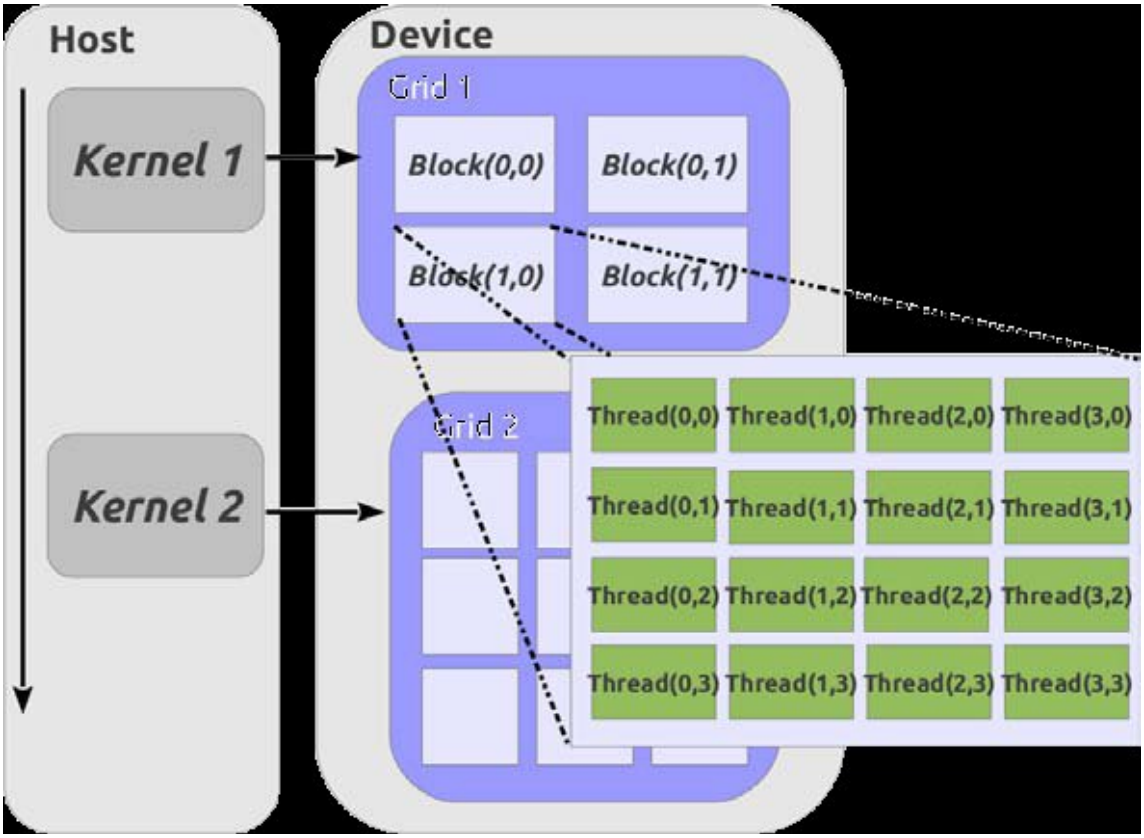


Figure II-1. Décomposition des données en grille de blocs, ou chaque bloc contient un certain nombre d'éléments exécutés en parallèle.

Le terme Host définit le CPU et le terme Device correspond au GPU. La dimension utilisée pour la définition de la grille est bidimensionnelle au niveau des blocs et des threads.

II.4-Organisation de la mémoire :

Chaque processeur, exécutant un thread, a un accès physique à plusieurs endroits mémoriels. Les deux plus larges zones mémoires sont les mémoires locales sur GPU et globale sur CPU, les processeurs du GPU ayant en effet accès à une partie de la mémoire du reste du système. Ces accès sont lents (entre 200 et 300 cycles d'horloge) et ne sont de préférence utilisés que lorsqu'ils peuvent être masqués par d'autres calculs.

Une partie de la mémoire globale du GPU peut être mise en cache sur un multiprocesseur, et accessible aux huit processeurs de ce multiprocesseur ; ce sont les caches pour les constantes et pour les unités de texture, de 8Ko chacun, accessibles uniquement en lecture. Chaque multiprocesseur propose également une mémoire partagée entre ses processeurs, de 16Ko, permettant l'échange d'informations rapidement entre eux et économisant la bande passante. Cette mémoire n'est cependant partagée qu'entre les threads d'un même bloc. Enfin, chaque processeur dispose de quelques registres qui lui sont propres. La figure II-2 résume cette organisation des espaces mémoire utilisée en CUDA.

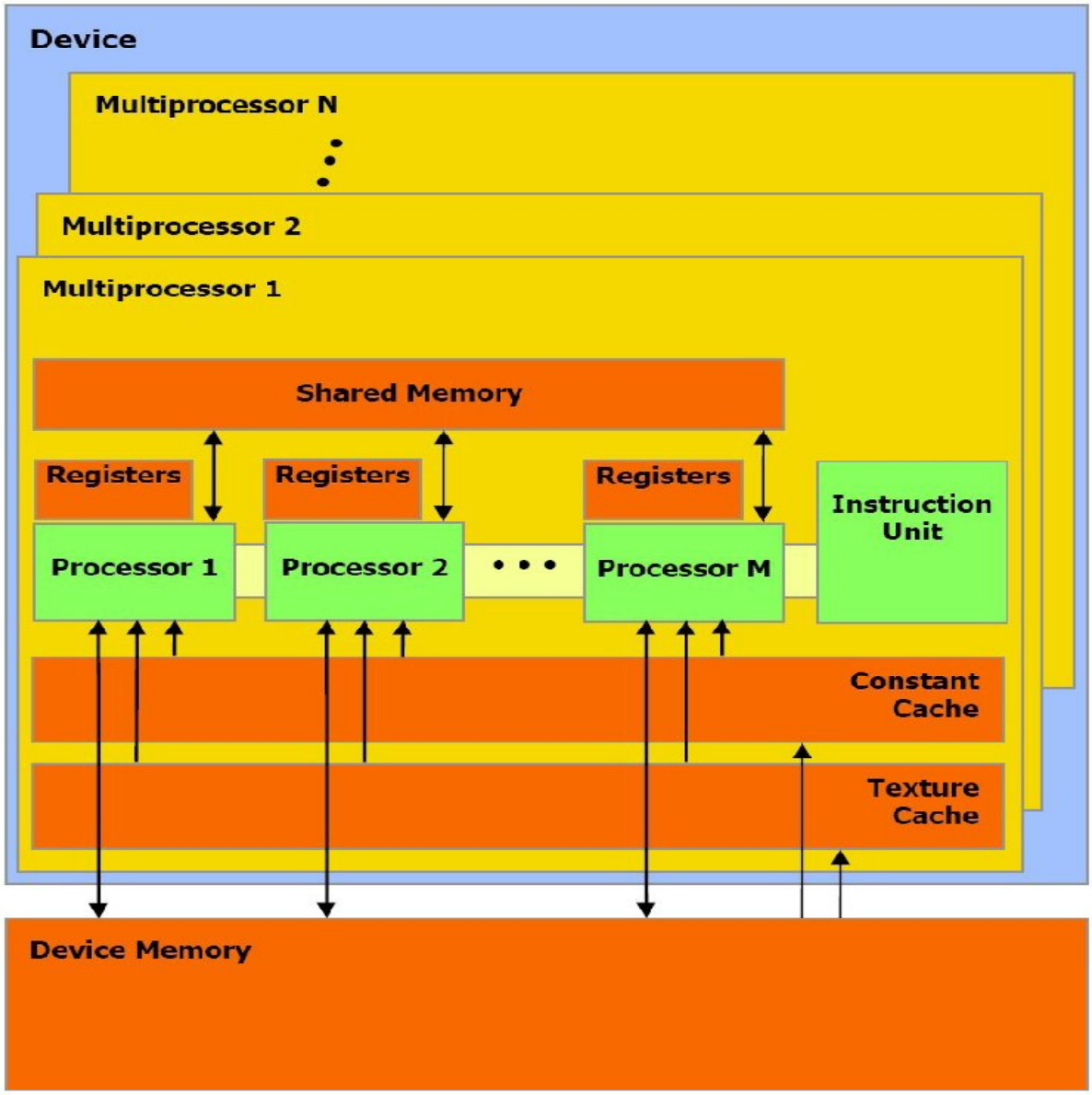


Figure II-2 Organisation de la mémoire

II.5-Architecture CUDA

A la différence des générations précédentes, qui partitionnaient les ressources de traitement en vertex et en pixel shaders, l'architecture CUDA comprend un traitement unifie permettant ainsi à chaque unité arithmétique et logique (ALU) du circuit d'être gérée par un programme afin de réaliser des traitements généraux. NVIDIA ayant souhaite que cette nouvelle famille de processeurs graphiques soit utilisée pour des traitements généraux, ces ALU implémentent la norme IEEE pour l'arithmétique virgule flottante simple précision et ont été conçues pour

utiliser un jeu d'instructions adapte à ces traitements généraux au lieu d'être spécialisées dans les traitements graphiques.

En outre, les unités d'exécution du GPU peuvent désormais lire et écrire n'importe ou en mémoire et également accéder a un cache (géré de façon logicielle) appelé mémoire partagée. Toutes ces fonctionnalistes de l'architecture CUDA ont été ajoutées afin de créer un GPU qui excelle dans les traitements généraux, avec de bonnes performances pour les traitements graphiques classiques.

II.6-Utilisation de l'architecture CUDA

L'effort de NVIDIA pour offrir au public un produit permettant a la fois d'effectuer des calculs et des traitements graphiques ne pouvait pas s'arrêter a la production d'un circuit intégrant l'architecture CUDA. Quel que soit le nombre de fonctionnalités ajoutées par NVIDIA a ses composants pour faciliter les traitements, il n'y avait toujours pas moyen d'accéder a celles-ci autrement qu'en passant par OpenGL ou DirectX. Ceci impliquait non seulement que les utilisateurs devaient continuer de traduire leurs traitements eproblèmes graphiques, mais également qu'ils devaient encore écrire ces traitements dans un langage de shading comme GLSL d'OpenGL ouHLSL de Microsoft.

Pour intéresser le maximum de développeurs possibles, NVIDIA a donc choisi le langage C – une norme incontournable de l'industrie informatique – et lui a ajoute un nombre relativement restreint de mots-clés afin d'exploiter certaines des fonctionnalités spéciales de l'architecture CUDA. Quelques mois après le lancement de la GeForce 8800 GTX, NVIDIA mit a disposition un compilateur pour ce langage, CUDA C, qui devint le premier langage conçu spécifiquement par un constructeur de GPU pour faciliter le développement de traitements généraux sur ses circuits graphiques. Outre la création de ce langage pour écrire un code,

NVIDIA fournit également un pilote matériel permettant d'exploiter l'énorme puissance de calcul de l'architecture CUDA.

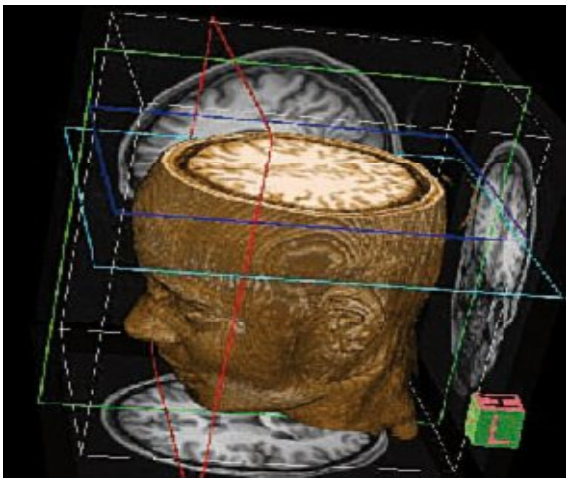
Désormais, les utilisateurs n'ont donc plus besoin d'apprendre les API Graphiques OpenGL ou DirectX ni de transformer leurs problèmes en traitements graphiques.

II.7-Applications de CUDA

Depuis son lancement au début de 2006, un grand nombre de sociétés et d'applications ont remporte un franc succès en choisissant de développer leurs applications en CUDA C.
Les avantages en termes de performances sont souvent de plusieurs ordres de grandeur par rapport a celles des implémentations traditionnelles précédentes. En outre, les applications s'exécutant sur les processeurs graphiques NVIDIA fournissent des performances supérieures par euro et par watt, comparées aux implémentations reposant exclusivement sur les technologies traditionnelles qui utilisent un processeur central. Les sections qui suivent représentent quelques exemples d'applications ayant su tirer parti de CUDA C et de l'architecture CUDA.

- **Imagerie médicale :**

  Grâce aux avancées technologiques et techniques, beaucoup de zones d'ombre du corps humain ont pu être éclairées, notamment, à travers l'utilisation de techniques d'imagerie médicale (radiologie, médecine nucléaire). Ces techniques permettent une reconstruction tridimensionnelle à très haute définition du corps humain permettant de le voir dans toutes ses dimensions et d'effectuer des coupes à divers endroits afin d'en détecter et d'analyser le moindre dysfonctionnement. D'un autre côté, le développement des techniques de visualisation informatique, d'analyse et des méthodes de reconstruction tomographiques 2D et 3D a créé de nouvelles opportunités pour la médecine. Ces techniques sont de plus en plus utilisées pour le planning d'opérations chirurgicales, la simulation de soins orthopédiques et orthodontiques, la simulation post opératoire et l'aide au diagnostic des maladies telles que le cancer. Ces applications impliquent l'utilisation d'algorithmes de plus en plus complexes tels que les algorithmes de segmentation d'images et de reconstruction d'objets 3D nécessitant la manipulation de grandes masses de données (data-intensive) et une grande puissance de calcul (CPU-time intensive) pour le traitement de ces données



FigureII.3-Image tomographique du cerveau en 3D permettant de faire des coupes sur tous les plans.

- **La modélisation du climat et les prédictions météorologiques**

La climatologie traite de la modélisation du climat et des océans afin d'en prédire et comprendre les bouleversements et les phénomènes naturels qui y sont liés. Elle permet de produire des prévisions météorologiques et de modéliser et simuler le déroulement des phénomènes et catastrophes naturelles telles que les inondations, les orages et les tempêtes de neige dans les pays à fort risque dont l'Algérie. Ces simulations permettent de prédire les quantités de pluies, épaisseur des neiges attendues et la vitesse des vents. De plus, les niveaux des crues des rivières et les niveaux d'eau dans les villes après de fortes précipitations peuvent être prédits en utilisant des modélisations mathématiques. Ces simulations et modélisations permettent de sauver des vies humaines en anticipant les aides, en déclenchant des alertes à temps et en évacuant la population, si nécessaire. Ces modélisations sont basées sur des modèles mathématiques et des systèmes numériques impliquant de nombreux paramètres. Ces modèles sont de leur part basés sur des lois de la physique (équations de thermodynamique et écoulement des fluides). La zone géographique à couvrir est représentée sous forme d'une grille dans laquelle sont disposés les paramètres initiaux de l'atmosphère récoltés à divers endroits par des stations météo spécialisées (température, pression, humidité et salinité des eaux de mers). Le modèle calcule alors l'évolution des paramètres à chaque noeud de la grille à des intervalles de temps bien déterminés. La précision des prévisions dépend de granularité

des mailles de la grille. Actuellement, les maillages utilisés sont généralement de l'ordre de 7 à 10 km selon les organisations.

Figure II.4-Simulation d'une inondation due à une crue d'une rivière après de fortes précipitations

- **Biotechnologies et Industrie pharmaceutique**

Le domaine des biotechnologies est actuellement en plein développement.

Des techniques de simulation et de modélisation de molécules sont utilisées dans de nombreux procédés en industrie pharmaceutique : la prédiction de structures de protéines inconnues (protein folding), la modélisation des interactions intermoléculaires et la conception de nouvelles molécules pour la découverte de nouveaux médicaments. La modélisation moléculaire se réfère à des modèles théoriques et techniques de calcul utilisés pour la représentation de structures de molécules et la simulation de leurs comportements et interactions. Ces techniques nécessitent d'une part le traitement de grands volumes de données durant et après les expérimentations, et d'autre part, l'utilisation d'une puissance de calcul considérable pour la modélisation des macromolécules complexes qui contiennent des milliers d'atomes constituant les protéines et les acides nucléiques (exp. ADN).

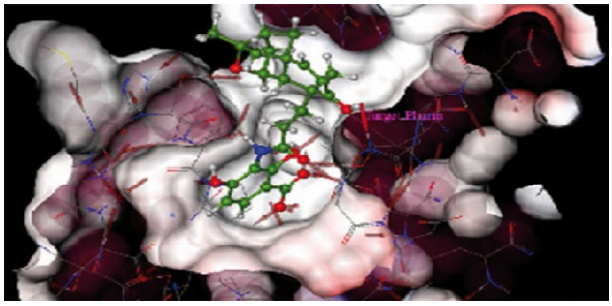


Figure II.5- Modélisation des macromolécules complexes qui contiennent des milliers d'atomes constituant les protéines et les acides nucléiques

- **Industrie pétrolière**

L'industrie pétrolière est l'un des domaines ayant un besoin énorme en puissance de calcul. Dans cette industrie, des simulations, des techniques d'optimisation et d'aide à la décision, de plus en plus complexes et gourmandes en puissance de calcul, sont utilisées pour la production et la recherche de nouvelles réserves énergétiques. Parmi ces techniques, nous citons les plus importantes : la modélisation et la simulation de réservoir, le traitement d'images 3D et l'interprétation de données provenant d'algorithmes sismiques. La simulation de réservoir est une technique utilisée afin de déterminer le plan de développement d'un gisement pétrolier. Les ingénieurs ont besoin de simuler l'historique de production d'un gisement en fonction de divers scénarios d'implantation de puits. Ces simulations passent par la résolution d'équations complexes modélisant l'écoulement des fluides dans les réservoirs. De plus, afin d'obtenir un aperçu détaillé des sites de forage, les compagnies pétrolières ont recours à des techniques d'imagerie sismique en 3D.

Le recours aux infrastructures de calcul haut performance est donc un besoin primordial pour les acteurs de ce domaine.

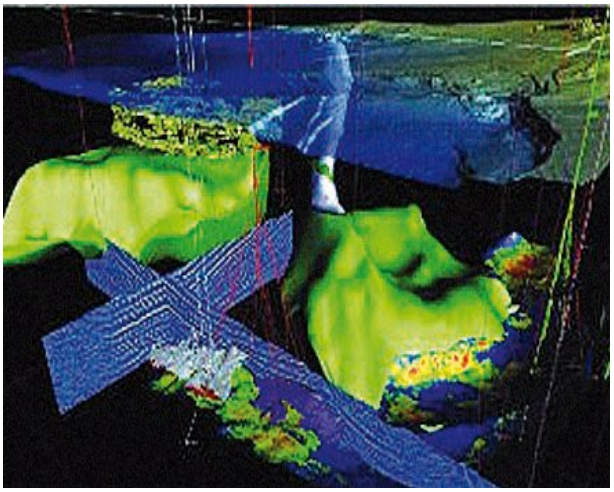


Figure II.6-Simulation d'un réservoir de pétrole. Les zones en vert clair représentent les réservoirs de pétrole.

- **Protection de l'environnement**

L'industrialisation forcenée de l'économie mondiale a eu pour conséquence naturelle de poser le problème de l'environnement auprès des populations. Les conséquences du changement climatique, la spirale infernale du prix du pétrole et la présence toujours plus grande de polluants dans l'air et l'eau ont mis en évidence les dommages collatéraux du progrès industriel. Malgré les problèmes qu'ils posent a notre environnement, les détergents et les produits de nettoyage sont depuis longtemps des produits de consommation dont on a besoin dans la vie quotidienne. De nombreux scientifiques ont donc commence a rechercher des moyens de réduire l'impact environnemental de ces détergents sans pour autant diminuer leur efficacité. Cependant, obtenir quelque chose sans contrepartie peut sembler illusoire.
Les composants essentiels des produits de nettoyage sont les *t*ensions actives.
Ce sont les molécules tensioactives qui déterminent la capacité de nettoyage et la texture des détergents et des shampooings, mais ce sont également elles qui font que ces produits ont un impact désastreux sur l'environnement. Ces molécules s'attachent a la saleté puis se mélangent a l'eau afin que les tensioactifs soient rinces en même temps que cette saletés.
Traditionnellement, la mesure de l'efficacité d'une nouvelle tension actif nécessitait de longs tests en laboratoire afin de déterminer les nombreuses combinaisons de matières et d'impuretés qu''ils pouvaient nettoyer. Ce processus, évidemment, pouvait être très long et très couteux.
L'université Temple de Philadelphie a travaille avec Procter & Gamble un acteur important du marche pour simuler les interactions des molécules tensioactives avec la saleté, l'eau et d'autres matières. L'introduction des simulations informatiques n'a pas servi uniquement a accélérer le travail en laboratoire traditionnel : elle a aussi permis d'étendre les tests aux nombreuses variantes de conditions environnementales, bien plus qu''il n'était possible de le faire avec les tests du passe. Les chercheurs de Temple se sont servis du logiciel de simulation HOOMD (*Highly Optimized Objectoriented Many-particle Dynamics*) sur deux GPU Tesla de NVIDIA, ce qui leur a permis d'obtenir des performances équivalentes de celles fournies par les 128 cœurs d'un Cray XT3 ou des 1 024 CPU de la machine BlueGene/L d'IBM. En augmentant le nombre de GPU Tesla, ils reussissent déjà a simuler les interactions des tensioactifs seize fois plus vite que les plates-formes précédentes.

CUDA ayant réduit de plusieurs semaines a quelques heures le temps de calcul de ce genre de simulation, les années a venir devraient donc voir apparaitre des produits nettoyants dont l'efficacité se sera accrue tout en ayant un impact moindre sur l'environnement.

II.8-Langages de programmation :

CUDA(v 6.0) : Computer Unified Device Architecture, est le dernier né des langages de NVidia, pouvant être utilisé à partir de la sixième génération de cartes graphiques de ce constructeur (architectures G80 et plus récentes). Quelques unes de ses particularités sont d'être encastré dans du code Cet C++ en définissant seulement quelques extensions, de mettre à disposition une mémoire partagée et rapide, ou de supporter différents types d'opérations scalaires, en particulier les opérations sur entiers et bit à bit. C'est aussi le premier langage à exploiter l'unification des shaders sur les architectures G80 de NVidia. Dans le domaine du traitement d'images, il est de plus en plus employé.

OpenCL : Le très récent OpenCL (Open Computing Language) a été annoncé par Apple au sein du Compute Working Group, formé par le Khronos Group (regroupant 3DLabs, Apple, AMD, NVidia, ARM, Ericsson et d'autres universitaires et industriels). Ces partenaires souhaitent mettre à disposition un langage open source, dans la veine d'OpenGL et OpenAL et ont pour ambition de faire d'OpenCL le standard libre pour le calcul GPGPU, avec les importants avantages d'être multiplateforme (cartes AMD/ATI et NVidia) et de permettre une programmation homogène

BrookGPU : Brook+ BrookGPU est une implémentation GPU du langage Brook, tous deux développés par le Stanford University Graphics Lab. C'est un langage basé sur la gestion de flux de données. AMD/ATI a également proposé Brook+, une amélioration de BrookGPU pouvant être utilisé uniquement sur leurs cartes. Folding@home, projet mondial de calcul distribué simulant les repliements de protéines dans le but d'en tirer
des solutions médicales, utilise en partie Brook+. **Scout** Proposé par le Los Alamos National Laboratory, Scout est un langage GPGPU destiné à l'analyse et à la visualisation scientifique, dont beaucoup de techniques sont basées sur l'utilisation de mappings, exprimés sous forme de fonctions mathématiques et transformant des données en image affichables. Scout a notablement été utilisé pour la simulation et la détermination de caractéristiques du courant côtier El Ninõ.

Accelerator : Microsoft Research a présenté Accelerator,destiné à simplifier la programmation GPGPU en fournissant un modèle de calcul parallèle accessible simplement à travers d'autres langages. Les opérations parallèles y sont compilées à la volée et optimisées pour les fragment shaders.

CGis : Le langage CGiS , développé par l'Université des Saarlandes, est un autre langage parallèle, similaire à Brook et Accelerator, manipulant également des objets de type flux de données, mais se démarquant par l'absence de notion de kernel computationnel (équivalent de shader pour du calcul uniquement GPGPU), remplacé par un mécanisme de boucle globale forall, chère au calcul parallèle.

II.9-Discussion :

Dans ce chapitre nous avons présenté CUDA, en commençant par décrire cet environnement de programmation suivi de son architecture avec son organisation mémoire. Après avoir présenté quelques exemples d'application dans déférentes domaines, nous avons cité les langages de programmation existant.

III.1-Préambule :

Dans ce chapitre on donnera les différentes configurations du matériel installé (Micro ordinateur, carte graphique) et les composants utilisés pour l'installation (Cuda) avec leurs différentes versions.

III.2-Description et installation matérielle :

a- L'ordinateur utilisé pour ce travail est un intel ® core ™ i3 -2120 dont la fréquence de CPU est de 3,30 GHZ (voir la figure 3.1)

Informations système générales

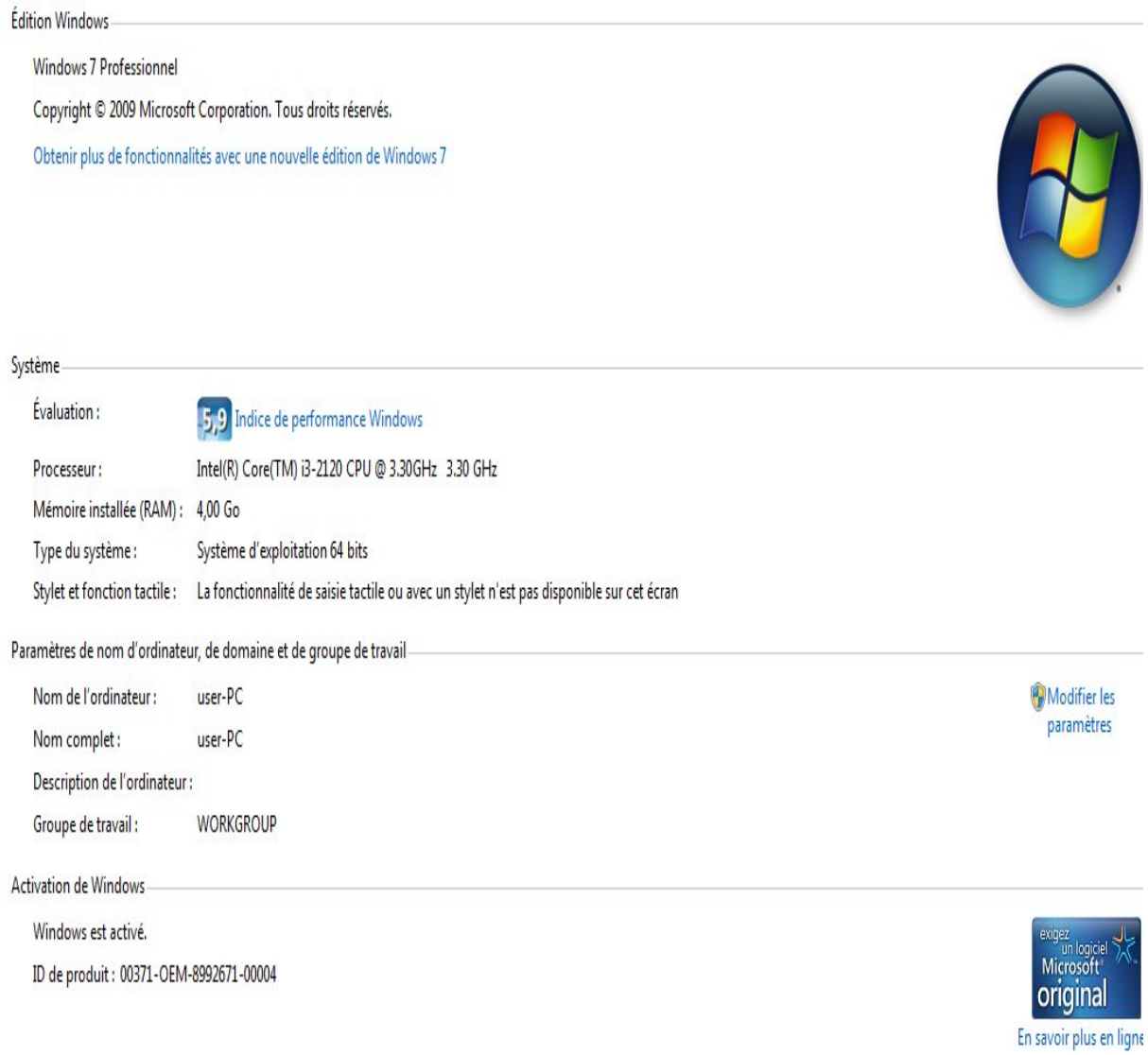


Figure III-1 Informations système générales

b-La carte graphique est une GTX650 (DDR5avec 1 GO) dont les caractéristiques sont données par le tableau suivant :

	HD 7770 GDDR5	HD 7750 GDDR5	GTX 650 GDDR5	GT 640 DDR3	GT 640 OEM GDDR5
GPU	Cape verde	Cape verde	GK107	GK107	GK107
Procédé de fabrication	28nm	28nm	28nm	28nm	28nm
Unités de calcul	640 scalar	512 scalar	384	384	384
Unités de texturing	40	32	32	32	32
ROPs	16	16	16	16	16
Fréq.GPU(MHZ)	1000	800	1058	900	950
Débit de triangles (Mtris/s)	1000	800	1058	900	950
Fillrate(Gpixels/s)	16.0	12,8	8,4	7,2	7,6
Débit de MADs SP (Gflops)	1280	819	813	691	730
Débit de filtrage (Gtexels/s)	40.0	25,6	33,9	28,8	30,4
Fréq.mémoire(MHZ)	1125	1125	1250	891	1250
Taille mémoire(Mo)	1024	1024	2048/1024	2048/1024	2048/1024
Bus mémoire(bits)	128	128	128	128	128
BP mémoire(Gio/s)	67.1	67,1	74,5	26,6	74,5
Direct3D	11.1	11,14	11,1	11	11

Tableau III.1 Caractéristiques de la carte graphique GTX650

C- Les composants utilisés :

En informatique les versions des logiciels évoluent très vite. Il est donc important de préciser les outils que nous utilisons ainsi que leurs versions :

- Windows 7 pro x64
- Carte graphique Nvidia GTX650
- Visual Studio 2012 Pro Edition
- NVIDIA Driver 9.18.13.32.88 for Windows 7 (64-bit) with CUDA Support
- CUDA_6.0.37_winvista_win7_general_64
- Toolkit 6.37 for Windows 7 (64-bit)
- CUDA SDK 5.5 code samples for Windows 7 (64-bit)
- Code Bloks 13.12

III.3-Installation de la plate forme CUDA:

Pour commencer il faut installer les **packages** fournis par Nvidia. Ils permettent à Windows de reconnaitre votre carte (si ce n'est pas le cas). Il fournit un ensemble d'outils (debugger sur gpu, accès direct dans la mémoire système,...) d'aide au développement sous CUDA. Et ils fournissent aussi un ensemble de programmes "vitrines" pour vous montrer la puissance de cette nouvelle technologie.

a -CUDA Toolkit: ensemble de logiciels de développement CUDA qu'on peut télécharger sur le site http://developer.nvidia.com/object/gpucomputing.html.

Cliquer sur le lien CUDA toolkit pour arriver sur la page de téléchargement de la figure NIII.2

Il vous sera demandé de choisir votre plate-forme parmi les versions 32 et 64 bits de Windows vista, Windows7, Linux et Mac OS. Puis exécuter le téléchargement

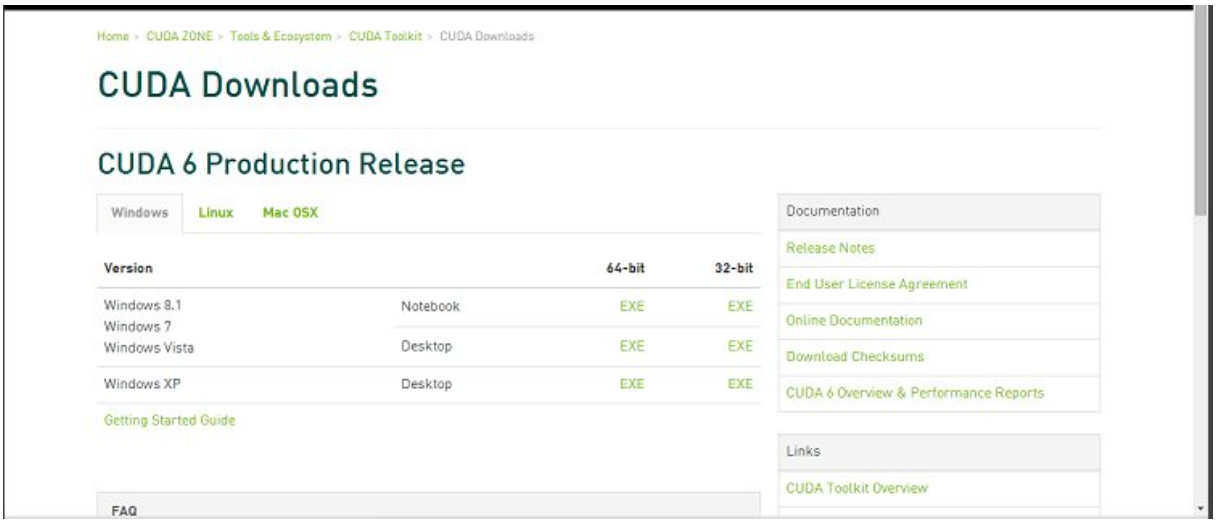


Figure III-2 CUDA Toolkit

b- CUDA SDK 5.5 code samples

Le paquetage GPU computing SDK code samples, contient des dizaines d'exemples de programmes à compiler et à exécuter.

CUDA samples est très important dans la mesure où nous utilisons le projet template fournit, pour créer de nouveaux projets CUDA.(figure III. 3).

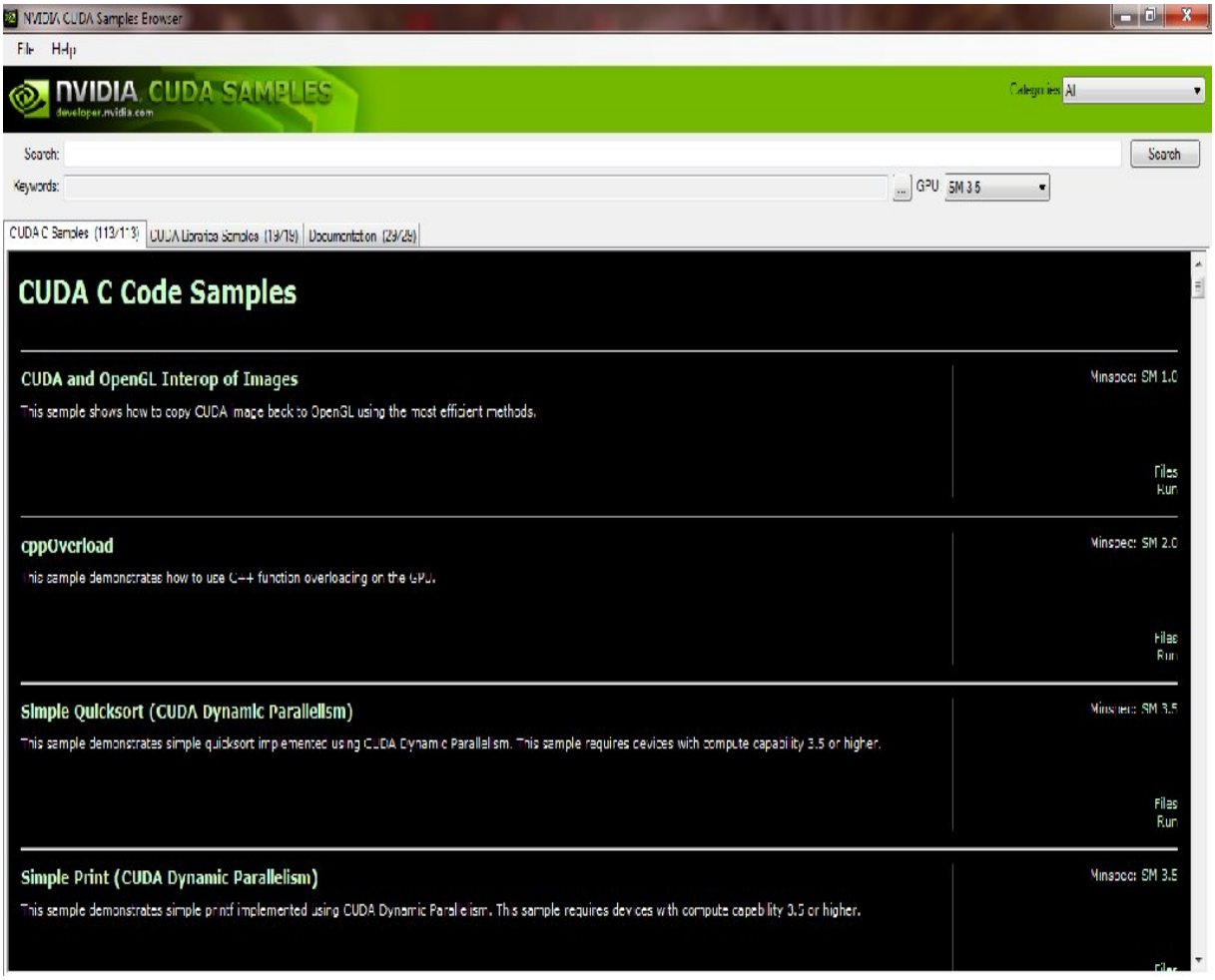
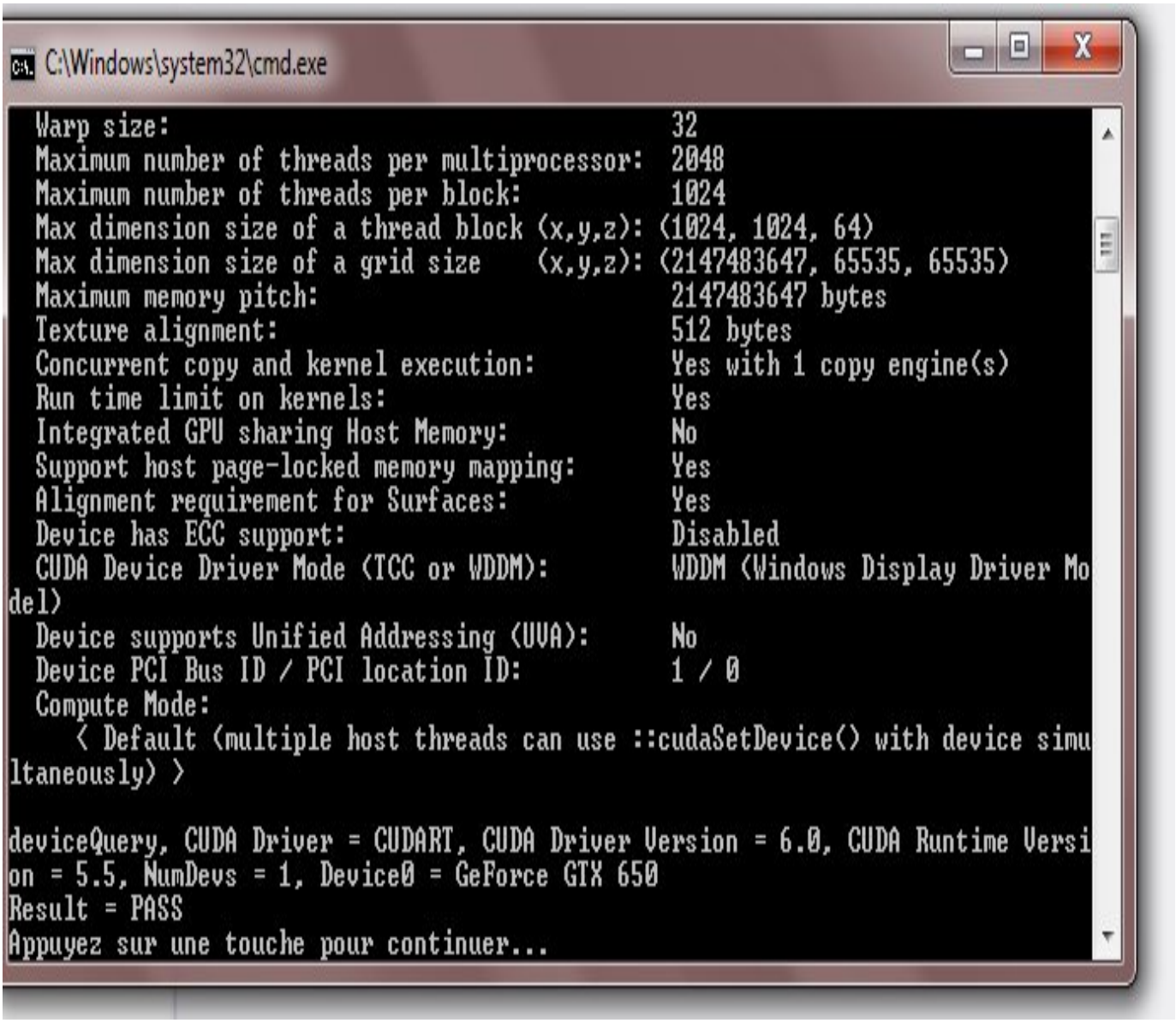


Figure III.3- CUDA SDK code samples

Pour tester l'installation de notre plate forme ,le projet deviceQuery nous permet de connaitre l'architecture de la notre carte graphique . La figure III.6 montre le résultats de son exécution.



La figure III.4 DeviceQuery

III.4 Les qualificateurs de fonctions :

Les qualificateurs de fonctions indiquent comment une fonction est appelée ou exécutée, suivant par exemple la syntaxe : __qualificateur__ void ma_fonction (arg_1, arg_2)

Il existe 3 qualificateurs :

– __host__, une fonction appelée par le CPU et exécutée sur le CPU, c'est l'option par défaut lorsqu'on ne met rien.

– __global__, une fonction appelée par le CPU et exécutée sur le GPU

– __device__, une fonction appelée par le GPU et exécutée sur le GPU

On peut combiner __host__ et __device__, dans ce cas la fonction peut être exécutée sur le CPU et sur le GPU.

III.4- Les variables

Les qualificateurs de variables indiquent le type de mémoire qui stocke la variable, suivant par exemple la syntaxe : __qualificateur__ int ma_variable ;

Il existe 3 qualificateurs :

– __device__, la variable est stockée dans la mémoire globale du GPU, elle est accessible par tous les threads

– __constant__, la variable est stockée dans la mémoire constante, elle est accessible par tous les threads

– __shared__, la variable est stockée dans la mémoire partagée, elle est accessible seulement aux threads du même block

On peut combiner __device__ avec les deux autres qualificateurs, leurs propriétés s'additionnent.

III.5-Tests et exemples sur CUDA :

III-5-1 Addition de deux tableaux :

Ce projet est séparé en deux parties, une première partie qui est du code C classique faisant l'addition de deux vecteurs codés sous forme de tableaux (TableA et TableB) vers un troisième tableau (Table C). La deuxième partie fait la même opération mais en parallélisant les calculs pour chaque cellule du vecteur.

Les différentes étapes pour paralléliser en Cuda sont :

– Création et allocation des tableaux sur le GPU
– Copier les données de la mémoire "CPU" vers la mémoire "GPU"
– Déclaration des threads
– Exécution des calculs en parallèle
– Récupération des résultats

La figure suivante illustre les résultats obtenus :

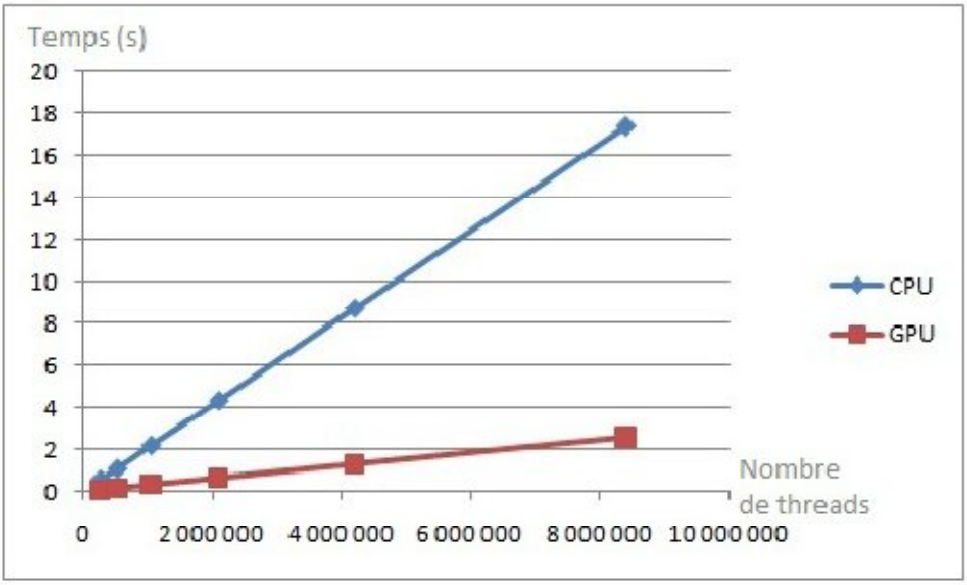


Figure III. 5– Graphe du temps (en seconde) d'addition de deux tableaux en fonction de la taille des tableaux.

D'après le graphe on voit que plus le nombre de threads est grand pour le GPU moins est le temps d'exécution d'où l'utilité de la plate forme.

III-5-2 Traitement de signal :

CUDA samples, nous livre une librairie assez riche pour des applications en traitement de signale ,les codes sources pour la FFT, le produit de convolution …etc, comme l'illustre la figure suivante :

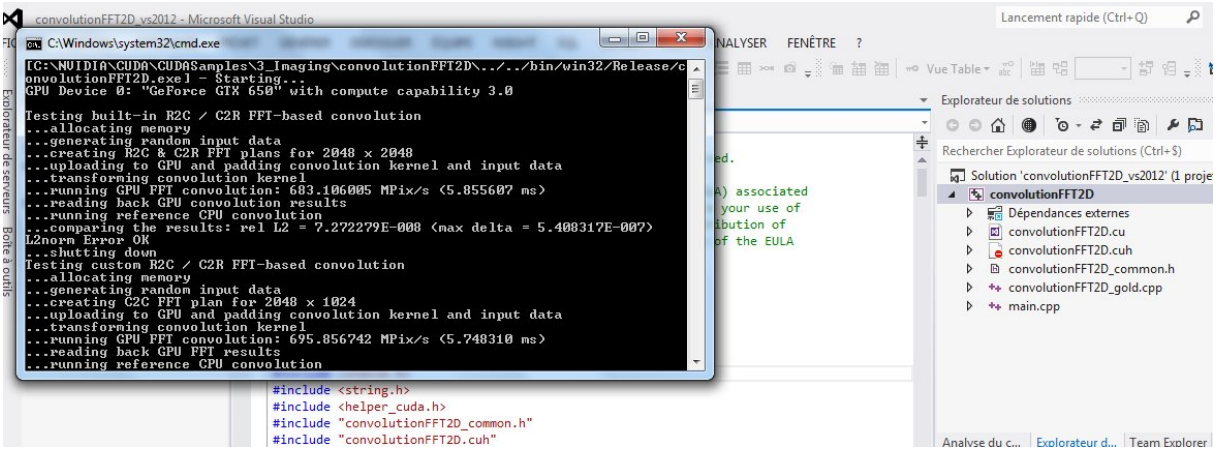
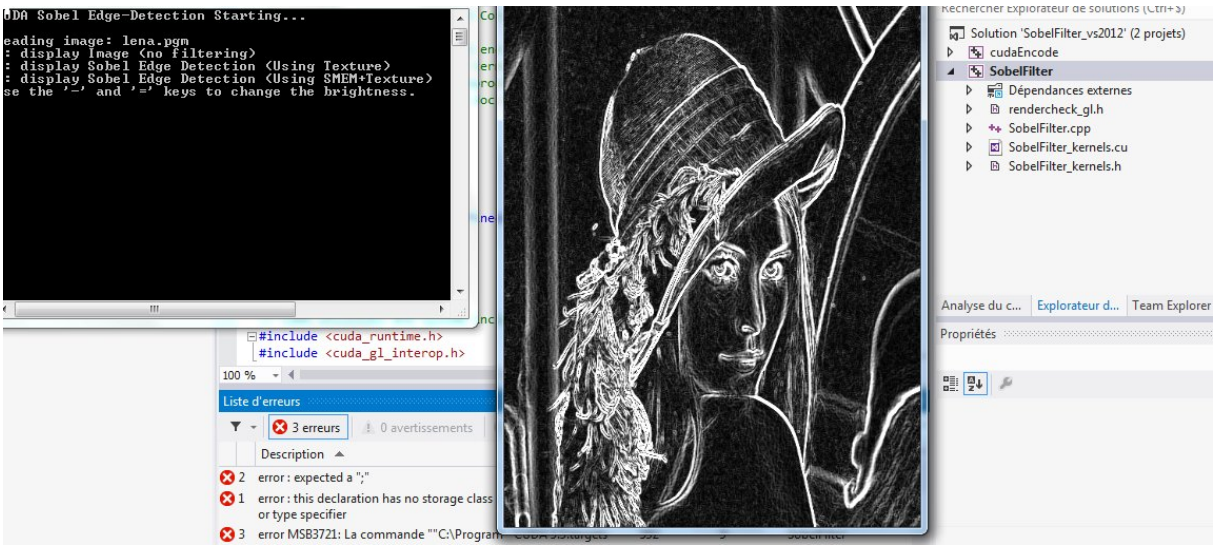


Figure III.6-Produit de convolution de la FFT à deux dimensions

III-5-3 Traitement d'image :

Il existe différentes implémentation sur GPU en traitement d'images, les boites à filtres proposée par CUDA est un exemple de parallélisassions d'algorithmes, comme le filtre de SOBEL et le filtre de GAUSS que montrent les figures III.7 et III.8 comme suit :



La figure III.7 Filtre de SOBEL

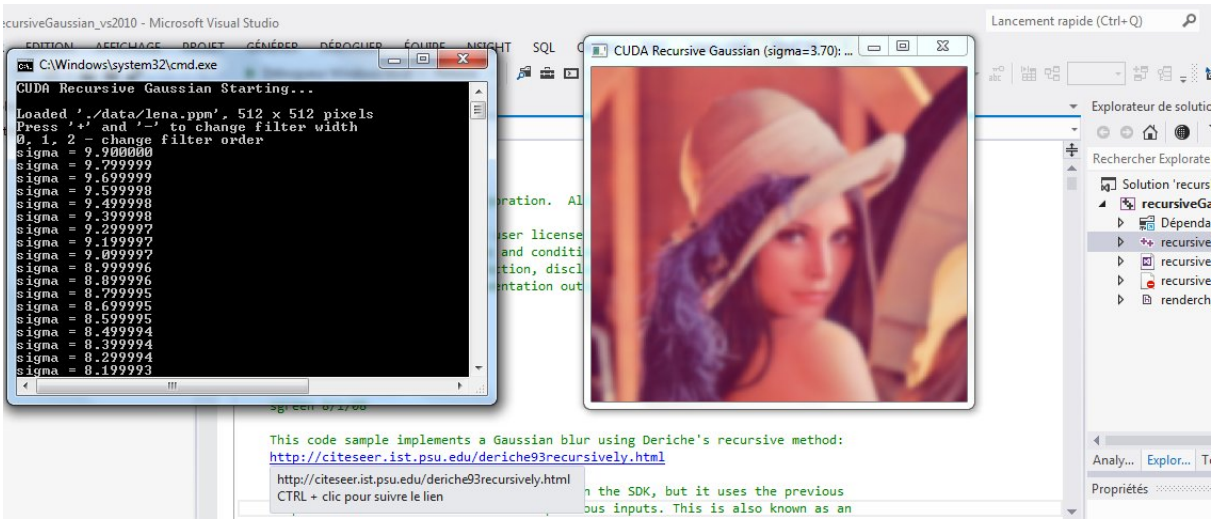


Figure III.8-Filtre de GAUSSE

Traitement graphique :

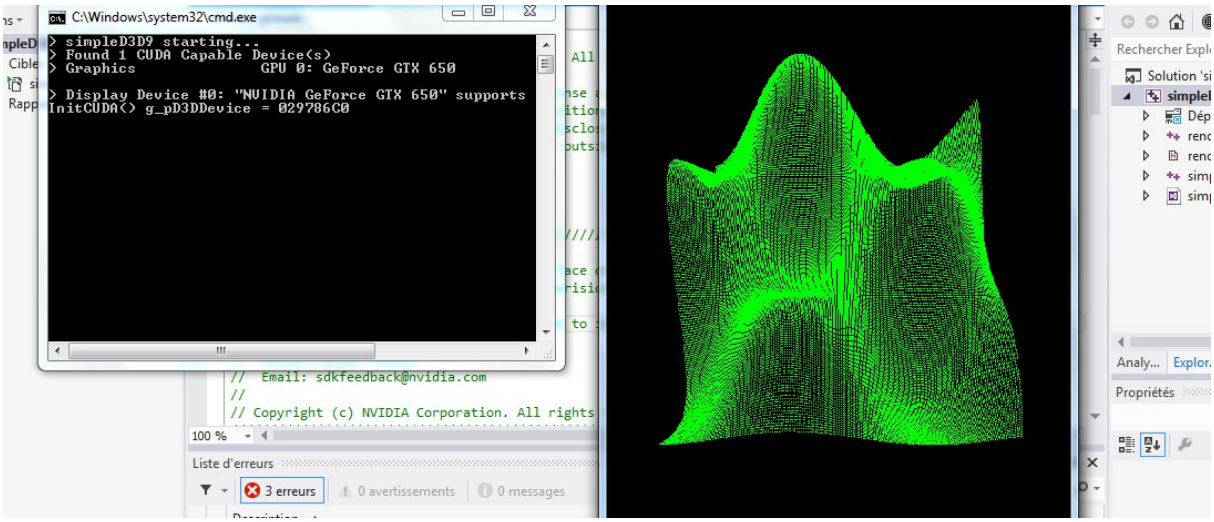


FIG III.9-Simulation : Exemple de mouvement de particules

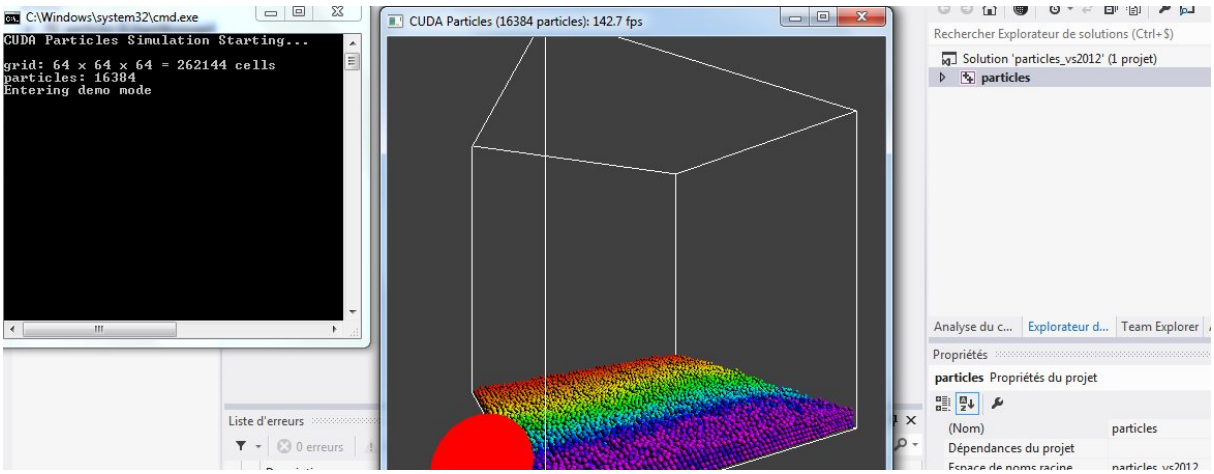


Figure III-10

III.5-Conclusion

Dans ce chapitre nous avons donné les différentes étapes d'installation de notre plate forme CUDA (matérielle et logiciel).La mise en marche des différents exemples (une centaine), livrés par CUDA SAMPLES.

IV-1 Préambule :

Nous allons dans ce présent chapitre évalué notre logiciel par une série de tests sur quelques images et les résultats obtenus seront représentés sous forme de graphes commentés, mais aussi de voir comment l'architecture présentée dans le chapitre deux intervient dans les performances de calcul. Le but de ce chapitre est de donner un état de l'art sur les différentes implémentations parallèles existantes pour la compression d'image par JPEG.

IV.1- Compression d'image :

La compression d'une image numérique permet de réduire le nombre de bits représentant un pixel. Les deux principes qui interviennent pour atteindre cet objectif sont **:**

- La réduction de la redondance (n'introduise pas de perte).
- La représentation approximative de l'information contenue dans l'image (introduisant des pertes).

Le processus de compression seul n'a aucun intérêt si les données compressées ne peuvent être restaurées, l'opération de restauration des données est appelée décompression. La décompression agit sur l'image compressée et génère l'image originale ou presque selon la méthode utilisée. La figure suivantes détaille chaque étape de cet algorithme.

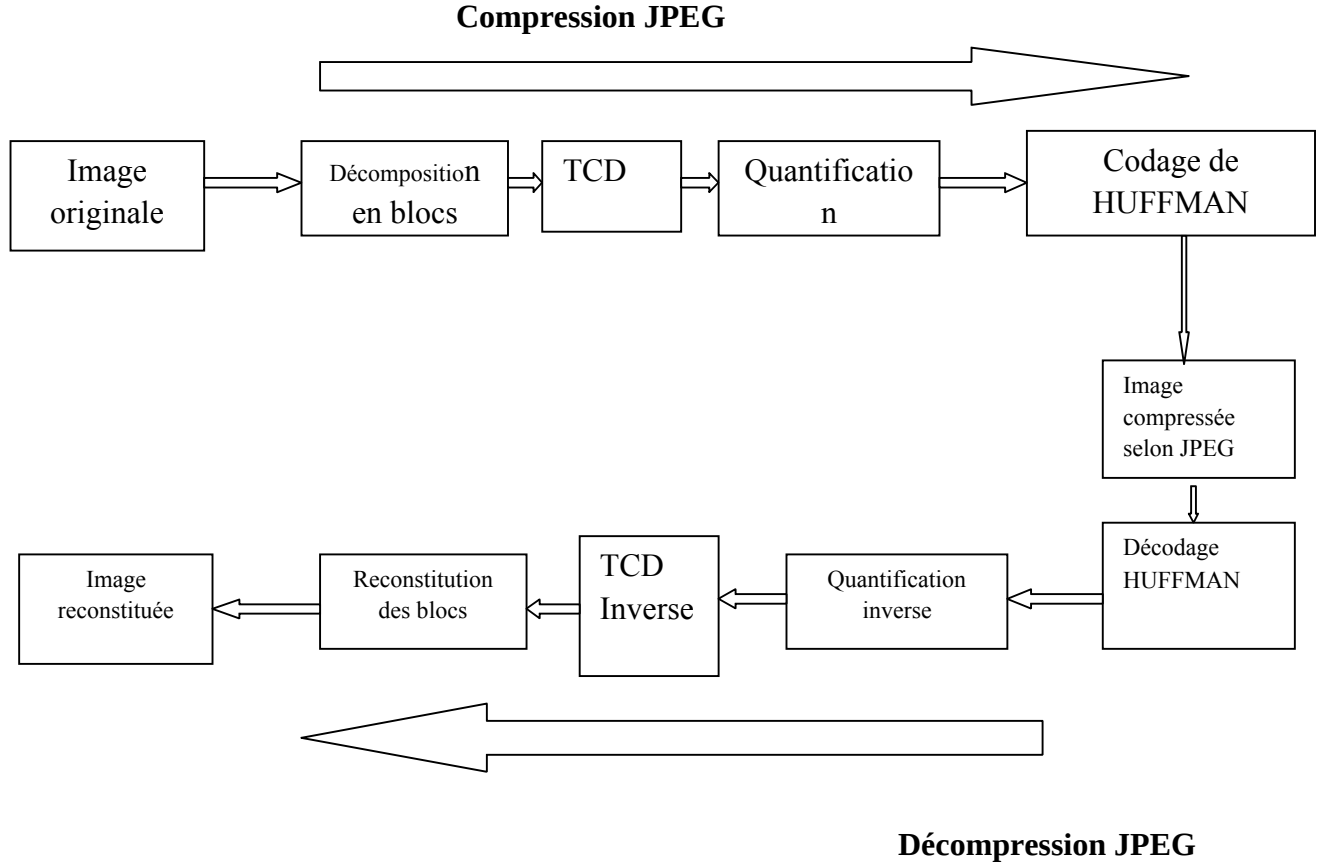


Figure IV -11 Etapes de compression et de décompression JPEG

a- Les différentes étapes du JPEG :

- **Décomposition en blocs** :

Le calcul de la **DCT** ne peut pas se faire sur une image entière parce que cela génère trop de calculs. Dès lors le groupe JPEG impose la décomposition de l'image en blocs de 8 pixels sur 8 pixels. La méthode de compression sera donc appliquée indépendamment sur chacun des blocs. Chaque bloc est représenté par un tableau de 64 nombres entiers compris entre 0 et 255.

- **Transformation DCT : transformée en cosinus discrète bidimensionnelle**

La clé du processus de compression est la DCT (Discrete Cosine Transform). La DCT est une transformée fort semblable à la FFT : la transformée de Fourier rapide (*Fast Fourier Transform*), travaillant sur un signal discret unidimensionnel. Elle prend un ensemble de points d'un domaine spatial et les transforme en une représentation équivalente dans le domaine fréquentiel. Dans le cas présent, nous allons opérer la DCT sur un signal en trois dimensions. En effet, le signal est une image graphique, les axes X et Y étant les deux dimensions de l'écran, et l'axe des Z reprenant l'amplitude du signal, la valeur du pixel en un point particulier de l'écran. La DCT transforme un signal d'amplitude (chaque valeur du signal représente l' "amplitude" d'un phénomène, ici la couleur) discret bidimensionnel en une information bidimensionnelle de "fréquence".

DCT :

$$(,) = -C(u).C(v)\sum \quad \sum \quad Img(x.y).\cos[-u +].\cos[- +].$$

- **La transformation inverse l'IDCT :**

$$Img(x.y)=-\sum \quad \sum \quad c(u).c(v).F(u,v).\cos[-u +].\cos[- +].$$

Où $\begin{cases} c(0)=(1)^{-1/2} \\ c(w)=1 \text{ pour } w=1,2,\ldots..N-1 \end{cases}$

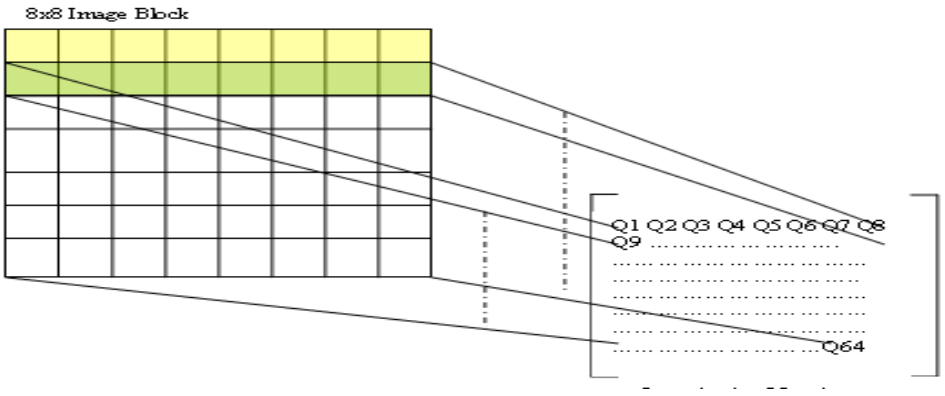
- **Quantification :**

Le but de la deuxième étape de la méthode JPEG, l'étape de quantification, est de diminuer la précision du stockage des entiers de la matrice DCT pour diminuer le nombre de bits occupés par chaque entier. C'est la seule partie non-conservative de la méthode (excepté les arrondis effectués). Puisque les informations de basses fréquences sont plus pertinentes que les informations de hautes fréquences, la diminution de précision doit être plus forte dans les hautes fréquences. La perte de précision va donc être de plus en plus grande lorsqu'on s'éloigne de la position (0,0). Pour cela on utilise une *matrice de quantification* contenant des entiers par lesquels seront divisées les valeurs de la matrice DCT. Ces entiers seront de plus en plus grands lorsqu'on s'éloigne de la position (0,0). Elle filtre les hautes fréquences.
Les matrices de quantifications intéressantes sont celles permettant de ``choisir'' la perte de qualité acceptable. Ce choix a été rendu possible grâce aux tests intensifs des matrices.

Habituellement, on prend pour matrice de quantification :

$Q=(q_{i,j})$ avec $q_{i,j}=$.( )

avec i l'indice de ligne, j l'indice de colonne et K le facteur de qualité (choisi entre 1 et 25).



Matrice de quantification

Codage de Huffman :

C’est un algorithme élaboré par David Albert Huffman en 1952,lors de sa thèse doctorat au MIT. Le but de cet algorithme est de réduire la redondance de la source en affectant des codes longs pour les symboles (pixels ou caractères par exemples) rares de des codes courts pour les symboles fréquents.

La construction du code de Huffman s’affecte de façon ascendante (des feuilles vers la racine de l’arbre).Sa mise en œuvre est décrite par l’algorithme suivant :

1-Le trie des symboles par ordre de probabilités décroissant (par exemple de la gauche vers la droite).

2-Successivement, les deux symboles de plus faibles probabilité d’apparition sont rattachés à un nœud dont le poids vaut la somme des probabilités des deux symboles. A ces derniers sont affectés respectivement les chiffres binaires « 0 » pour le plus fréquent et « 1 » pour le plus rare.

3-Le même procédé est répété est considérant les nœuds comme des symboles jusqu'à obtenir un seul nœud parent appelé racine.

4-Le code de chaque symbole s’obtient en écrivant de gauche vers la droite, les chiffres binaires rencontrés en parcourant l’arbre de la racine vers le symbole. Ainsi, plus le symbole est rare plus il est profond dans l’arbre, plus son mot de code est long.

Implémentation de module CUDA pour le JPEG :

Les tests sont effectués sur trois images de formats différents

Image 512MB

Image 1024x768

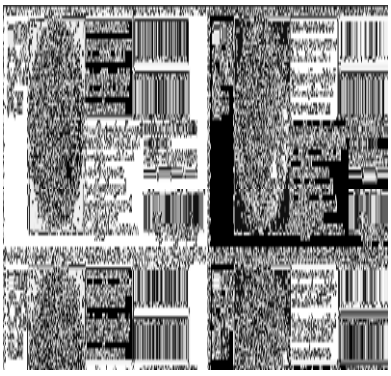
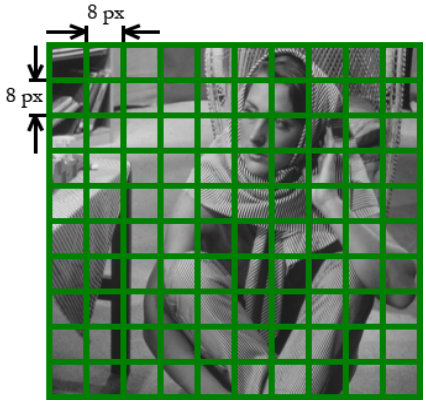


Image 2048x2048

Etape 1 :

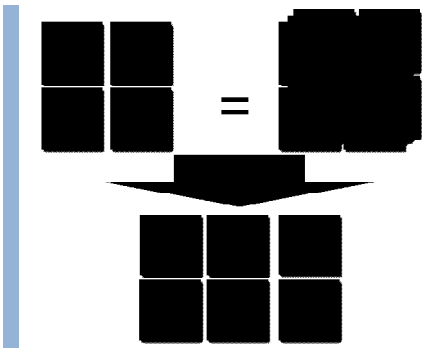
La compression commence par le chargement d'une image non compressé sous forme d'image en mode 24 bits à 8 bits rouge, vert et valeurs de bleu de chaque pixels.



Etape 2 : Application de la DCT

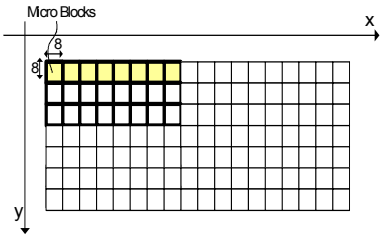
L'image est chargée dans la mémoire principale et le JPEG effectue des opérations sur les blocs de 8*8 pixels appelés MCU (unités minimales codée).Ces blocs facilitent les opérations de fréquences tel que la DCT .

La première étape après le chargement de l'image est de transformer les valeurs RVB à YCC (luma, chroma bleu,chroma rouge).Le but de la conversion est de séparer la luminosité (luminance)de l'image de couleurs différentes.

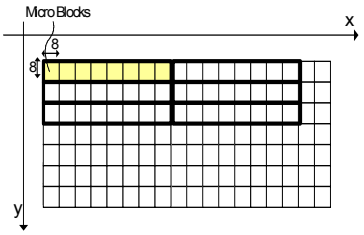


DCT (grille et blocs)

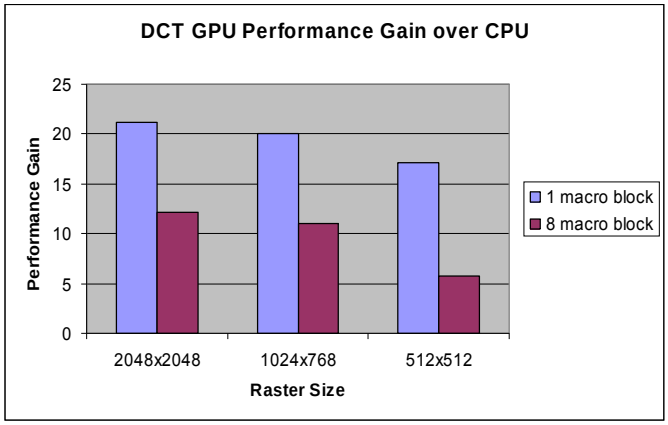
Méthode 1 :Chaque bloc est partagé en 64 threads(1 macro bloc)



Méthode 2 : Chaque bloc est partagé en 512 threads(8 macro bloc)



Résultats de le DCT :



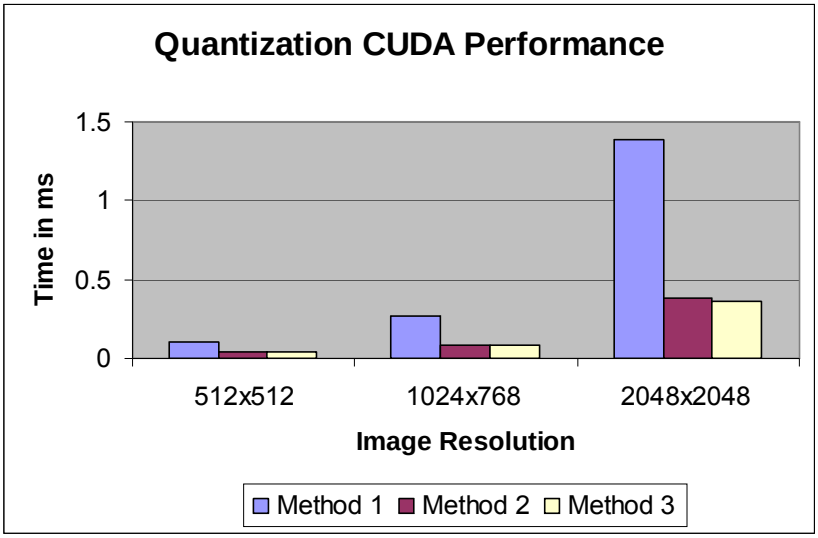
Etape 3 : Quantification

Méthode 1 - mise en œuvre exacte que dans CPU

Méthode 2 - copier l'image de 8x8 dans la mémoire partagée

Méthode 3 - Chargez les valeurs divisée en mémoire partagée.

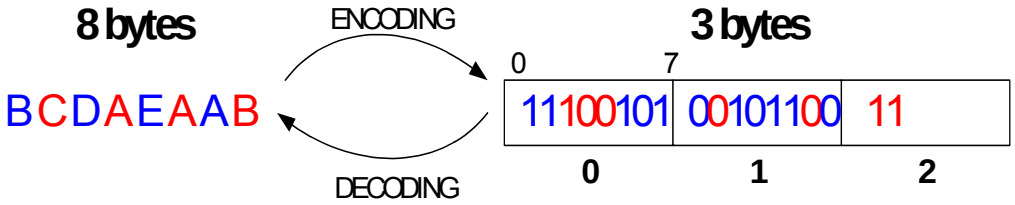
Résultats de la quantification ;



Méthode 2 et Méthode 3 ont des performances similaires sur les petites tailles d'image
Méthode 3 pourrait faire mieux sur les images plus que 2048x2048

Etape 4 : Codage de HUFFMAN :D'après les tests effectués sur les trois méthodes ,on a constaté que le parallélisme n'est pas applicable pour le codage de HUFFMAN.

Le codage et le décodage est un calcul périodique comme le montre la figure suivant



Temps d'exécution de la DCT et la IDCT :

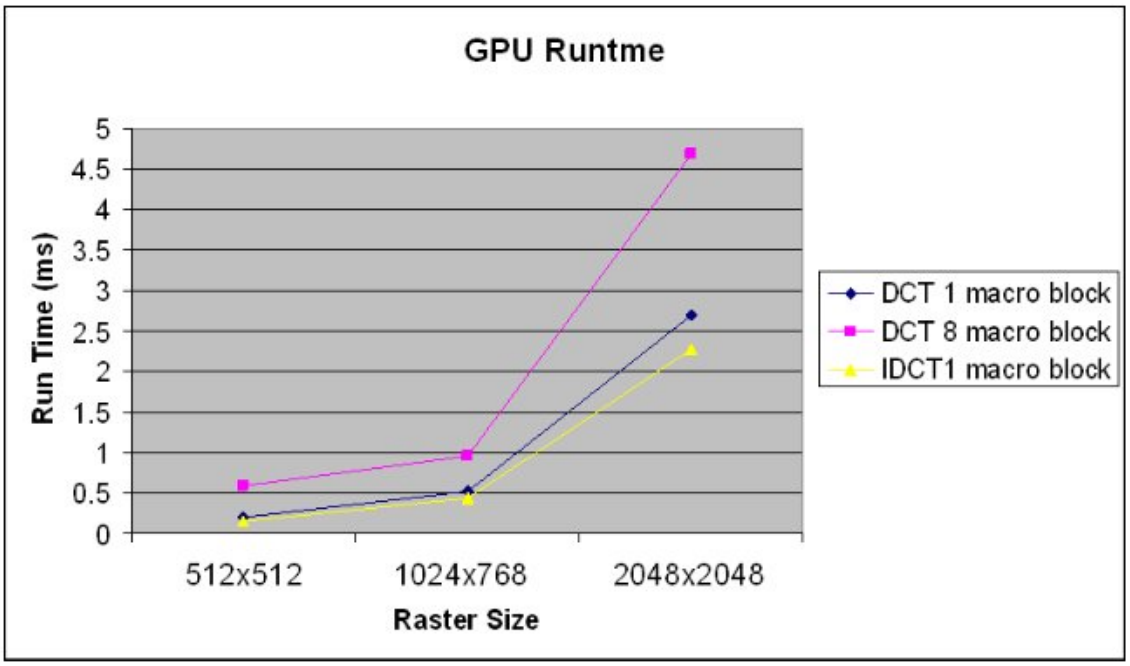


Figure I.3-Temps d'exécution pour la DCT et la DCT inverse pour les différentes implémentations du GPU

Temps/ms (CPU)	**512x512 - CPU**	**1024x768 - CPU**	**2048x2048 - CPU**
DCT	3.38	11.05	57.12
Quantization	5.74	17.16	75.97
IDCT	3.34	10.49	56.5
Temps/ms(GPU)	**512x512 - GPU**	**1024x768 - GPU**	**2048x2048 - GPU**
DCT	0.191	0.47	2.7
Quantization	0.039	0.085	0.379
IDCT	0.171	0.436	2.145

Performance Gain	**512x512**	**1024x768**	**2048x2048**
DCT	17.69633508	23.5106383	21.15555556
Quantization	147.1794872	201.8823529	200.4485488
IDCT	19.53216374	24.05963303	26.34032634

Tableau IV.1

Conclusion:

Dans ce chapitre nous avons présenté les tests d'une recherche [ ] sur l'application de la compression d'image par JPEG sur CUDA, les performances en gain permettent d'évaluer l'efficacité de la plate forme CUDA pour le traitement parallèle.

Conclusion

CUDA est une architecture qui facilite le développement de programmes parallèles. Associé à une plateforme logicielle complète, CUDA permet de tirer parti de la grande puissance des processeurs graphiques, ou GPUS, afin de construire des applications très performantes dans de multiples domaines – ingénierie, calcul, finance, etc ; et ce sans nécessiter de connaissances en programmation graphique : il suffit de savoir programmer dans une version légèrement modifiée du langage C.

Au cours de ce mémoire, nous nous sommes intéressés à l'utilisation générale des cartes graphiques au tant que machines de calcul parallèle.

Après avoir présenté les différentes générations des CPUS et GPUS, leurs architectures et l'organisation de leurs différents espaces mémoires ; nous avons donné les étapes d'installation de la plateforme CUDA. Quelques exemples livrés par CUDASAMPELS, montrent la diversité des domaines d'application de CUDA et du calcul parallèle.

Le but de notre travail est l'implémentation d'une plateforme CUDA pour la compression JPEG comme un état de l'art de calcul sur les GPUS NVIDIA.

Avec l'ouverture des interfaces de programmation (API) des GPUS, de nombreux chercheurs ont entrepris de les exploiter pour booster les traitements, habituellement destinés aux CPUS. Les calculs peuvent donc être portés sur GPU, mais le mode de fonctionnement SIMD de ces derniers impose la nécessite d'une grande quantité de données en entrée. En effet, ces processeurs tirent leurs forces du recouvrement des threads inactifs (en attente de données à distance du coeur) par l'exécution d'autres threads actifs (dont les données sont disponibles.

Nous l'avons vu, les constructeurs de processeurs graphiques ont fait évoluer leurs produits pour s'adapter aux contraintes de l'industrie. Pour sa part, *Nvidia* a établi plusieurs accords de partenariat avec quelques grands noms du logiciel industriel. C'est le cas de *National*

Instruments qui a apporté des modifications à LabView pour le support des GPUS. L'environnement de programmation graphique était déjà capable de générer du code pour des processeurs multicoeurs et des composants FPGA. Il donne désormais accès aux librairies CUDA pour déployer des applications sur des plateformes hétérogènes (CPU et GPU). Matlab offre également la possibilité d'effectuer des calculs sur GPU.

Pour conclure, nous souhaitons que le travail présenté dans ce mémoire sera un début pour Un sujet de recherche pour développement de programmes sur les plateformes parallèles .

Bibliographie

[1] http://fr.wikipedia.org/wiki/Loi_de_Moore

[2] Christophe RUBECK : Calcul hautes performances pour les formulations intégrales en électromagnétisme basses fréquences Intégration, compression matricielle par ondelettes et résolution sur architecture GPGPU thèse université de GRENOBLE.

[3] Alexandre Chariot ,Quelques Applications de la Programmation des Processeurs Graphiques à la Simulation Neuronale et à la Vision par Ordinateur thèse Ecole doctorat université de GRONOBLE DEC 2012

[4]ERIC CARION « Fonctionnement et performance des processeurs » université de pau et des pays del'adour

[5] Lilia Ziane el khodja « Résolution de systèmes linéaires et non linéaires creux sur grappes de GPUs thèse de l'Université de Franche Comté. »

[6] Implantation optimisée d'estimateurs de mouvement pour la compression vidéo sur plates-formes hétérogènes multi composants thèse doctorat à INSTITUT NATIONAL DES SCIENCES APPLIQU_EES DE RENNES

[7] Contributions aux techniques d'ordonnancement sur plates-formes parallèles ou distribuées université de franche -comté

[8]JONATHANC CLAUSTRE « Model particulaire 2D et 3D sur GPU pour plasma froid magnétisé ».Université de toulouse 3

[9] http://ppl.stanford.edu/cs315a/pub/Main/CS315a/CUDA_Compression_Final_Report.

[10] http :www.NVIDIA.COM/TESLA[11]Daniel Etumble « GPU et CUDA» cours de M1 informatique 2012 -2013[12]Anne sophie mourou « Introduction à la programmation des GPU »

[13] « CUDA BY EXAMPLES »LIVRE DE JASON SANDERS

[14]Application de la courbe d’hilbert dans la compression d’image fixes avec JPEG