Module SE301 – Rapport - année 2021/2022

Génération d'images et amélioration des performances

Étudiants :

Anouar Walziki
&
Wassim Mekki
&
Souhail Assal

Encadrants :

Jeremie Crenne
&
Bertrand Le Gal

Table des matières

1 Introduction

Afin de mettre en pratique certaines notions que nous avons apprises durant notre formation en tant qu'élèves ingénieur en systèmes embarqués. L'ENSEIRB-MATMECA nous propose un module de projet avancé. Dans ce module, l'objectif est de choisir un parmi les projets suggérés et de le réaliser pendant le 9 ème semestre.

Ce projet, nous permet non seulement d'investir et de mettre en œuvre nos compétences préliminaires, mais aussi, il nous présente une véritable occasion pour enrichir nos connaissances en termes d'électronique embarquée. Dans ce contexte, nous avons entrepris comme sujet : Génération d'images et optimisation des performances.

Notre sujet consiste alors, à faire tourner un projet C++ générant un ensemble de Mandelbrot sur des machines intégrant des processeurs Intel et des cartes graphiques NVIDIA, assurer la parallélisation des calculs en bénéficiant de l'architecture multi-coeurs de CPU, les ressources GPU en utilisant les bibliothèques OpenMP et CUDA. Enfin, nous réaliserons un comparatif de performances en exécutant le projet sur différentes machines. Dans une deuxième partie, on refait le même travail pour ray tracer.

Dans le présent rapport, nous allons résumer le travail que nous avons effectué et nous allons expliquer les différentes phases de réalisation de notre projet. Le rapport sera structuré comme suit : Dans le premier chapitre nous entamerons une étude bibliographique pour définir et rappeler quelques notions indispensables à savoir la génération d'image, parallélisation de calcul et les architectures CPU et GPU. Le deuxième chapitre sera réservé à la partie génération de Mandelbrot, son optimisation, et établissement d'un comparatif de performances. Le troisième chapitre concernera la partie de ray tracing. Enfin, nous clôturerons ce compte rendu par une conclusion générale.

2 État de l'art

2.1 Parallélisation des calculs

Le parallélisme en informatique est l'ensemble de techniques qui consistent à mettre en œuvre des architectures d'électronique numérique ainsi que des algorithmes spécialisés permettant de traiter des informations indépendantes de manière simultanée. Celui-ci, en exécutant une même tâche, partitionnée et adaptée, afin de pouvoir être répartie entre plusieurs processeurs ou entre plusieurs cœurs de calcul.

Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible. La réalisation de ces objectifs se traduit par des économies dans presque tous les domaines du calcul incluant, la dynamique des fluides, les prédictions météorologique, la modélisation et simulation de problèmes de dimensions plus grandes, l'intelligence artificielle, le traitement de l'information et l'exploration de données, le traitement et la génération des images...

En fait, à cause des contraintes technologiques qui ont limité l'augmentation des fréquences des processeurs (d'où leurs vitesses de calcul), les ingénieurs ont pensé à développer des architectures numériques parallèles qui sont devenues le modèle dominant pour tous les ordinateurs de nos jours. En effet, il s'agit des CPU multi-cœurs de calcul, traitant plusieurs instructions en

même temps au sein du même composant qui a pu résoudre ce dilemme pour certaines applications.

Cependant, pour d'autres applications et en l'occurrence celles qui concernent l'imagerie, la valeur ajoutée de ces processeurs génériques était médiocre. Ce qui a amené les chercheurs à inventer de nouvelles architectures appelées des GPU (Graphics Processing Unit) il s'agit des unités de traitement graphique. C'est en réalité une puce qui vise à optimiser le rendu d'images, l'affichage 2D et 3D, ou encore les vidéos. En effectuant les calculs mathématiques plus rapidement, il libère de la puissance de traitement pour le CPU qui peut alors se consacrer à d'autres tâches.

Le GPU est multitâche et il englobe un nombre de cœurs de calcul beaucoup plus important que les processeurs généraux, et capable de gérer une multitude de calculs en simultané, pour un rendu d'images très rapide.
Théoriquement le gain de temps d'exécution peut être expliqué des deux formules suivantes :

— Temps d'exécution d'un programme séquentiellement : $t_{exc} = nbr_{instructions} * t_{instruction}$
— Temps d'exécution d'un programme parallèle : $t_{exc} = nbr_{instructions} * t_{instruction}/nbr_{coeurs}$

Mais ce paramètre peut être influé par l'architecture et la façon d'accédé à la mémoire de donnée.
Selon la taxonomie de Flynn les programmes et les architectures sont classés selon le type d'organisation du flux de données et du flux d'instructions.

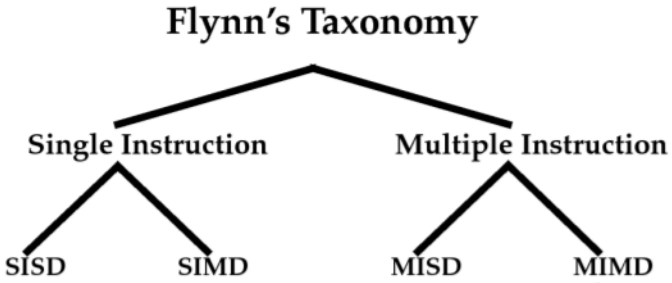

FIGURE 1 – La taxonomie de Flynn

La figure suivante représente les architectures SISD, SIMD, MISD, et MIMD.

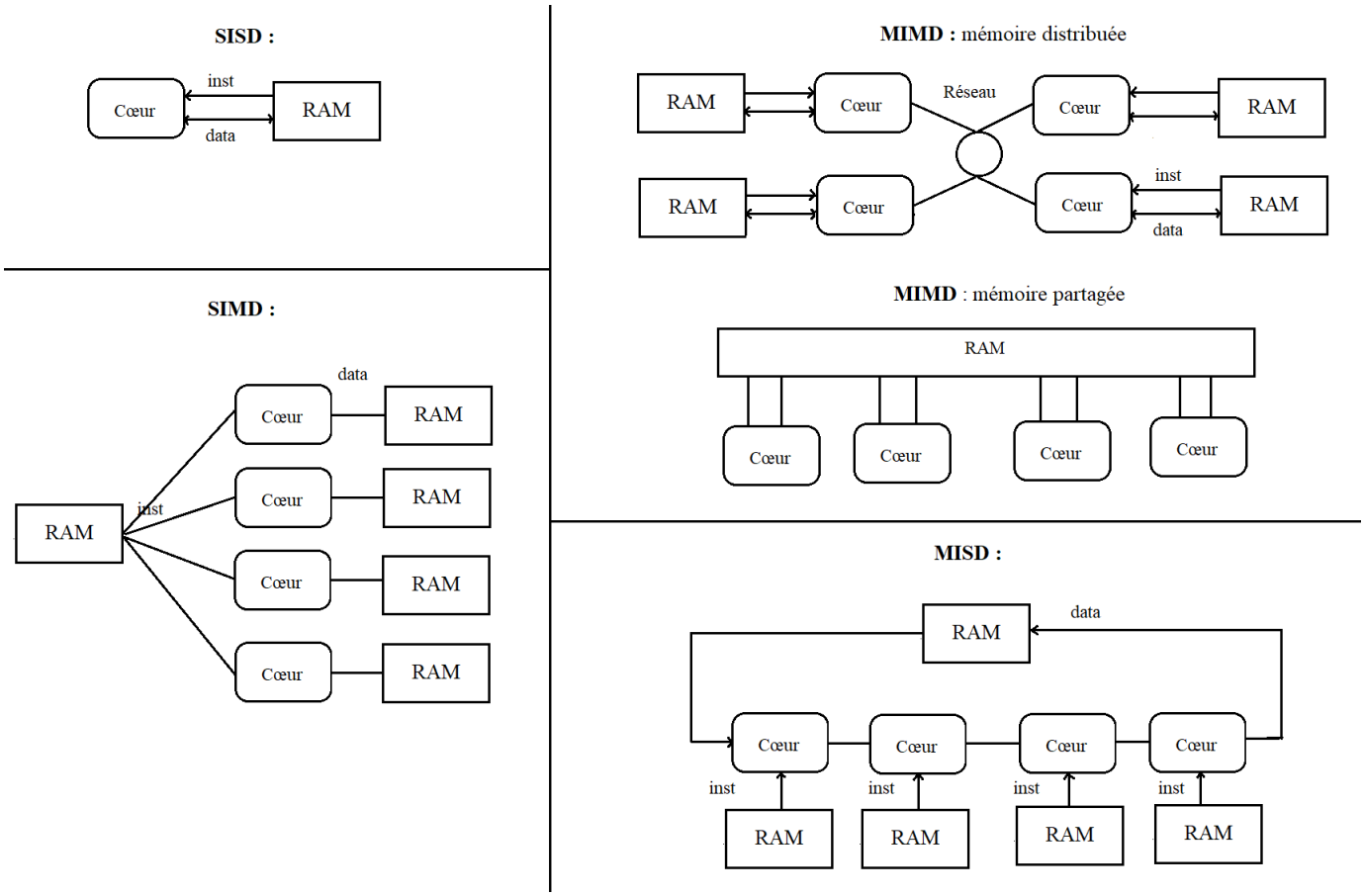

FIGURE 2 – Les architectures SISD, SIMD, MISD, et MIMD.

— SISD une seule unité de calcul, séquentielle accédant à une seule donnée à la fois.
— SIMD plusieurs unités de calcul, exécutant toutes la même instruction sur des jeux de données différents.
— MISD : plusieurs unités de calcul, qui exécutent des instructions différentes mais accédant à une seule donnée à la fois (fonctionnement de type pipeline)
— MIMD mémoire distribuée : plusieurs unités de calcul séquentielle, reliées entre elles par un réseau fonctionnement asynchrone, chaque cœur possède sa propre mémoire.
— MIMD mémoire partagée : plusieurs machines SISD, partageant une mémoire unique.

Avec ces architectures, il ne reste au développeur que l'optimisation de son code séquentiel avant de se lancer dans l'exploitation des différentes ressources CPU et GPU par l'intermédiaire des bibliothèques et des Framework spécifiques et de placer les instructions de parallélisation au bon endroit de son code de façon judicieuse afin d'améliorer les performances temporelles de son application son que celle-ci soit dégradée.

2.2 Comparaison des performances

L'évaluation des performances de calcul est un outil clé pour la recherche dans le domaine des architectures des processeurs (CPU) et processeurs graphiques (GPU). La croissante continue de la complexité de ce type d'architectures informatiques rend cette tache de plus en plus complexe. Le problème qui réside dans le développement de méthodes efficaces de évaluation de performances est de trouver le meilleur compromis entre précision et vitesse. Ce compromis

dépend de l'utilisation particulière du processus d'évaluation.

L'approche la plus courante pour estimer les performances des CPU et GPU réside dans la programmation de modèle logiciel servant à générer des images ou une séquence vidéo impliquant un nombre très important de calcul. Puis ensuite de créer un benchmark.

Un benchmark consiste en un banc d'essai qui a pour but de mesurer les performances. Il existe différente méthodes permettant de réaliser un banc d'essai pareil, la plus simple reste de calculer le temps d'exécution du logiciel à simuler. Une autre méthode classique lorsque le logiciel en question serve à des simulations vidéo est de calculer le nombre d'images affichées par seconde. C'est ce qu'on appelle de façon abrégée "fps" qui signifie frame per second en anglais, soit images par seconde en français (ips).

2.3 Architecture du CPU

L'architecture d'un CPU peut être regroupée selon trois parties majeurs : une unité arithmétique logique (ALU), une unité de contrôle (CU) et des registres. Cette architecture qu'on retrouve sur la figure suivante sera détaillée juste après.

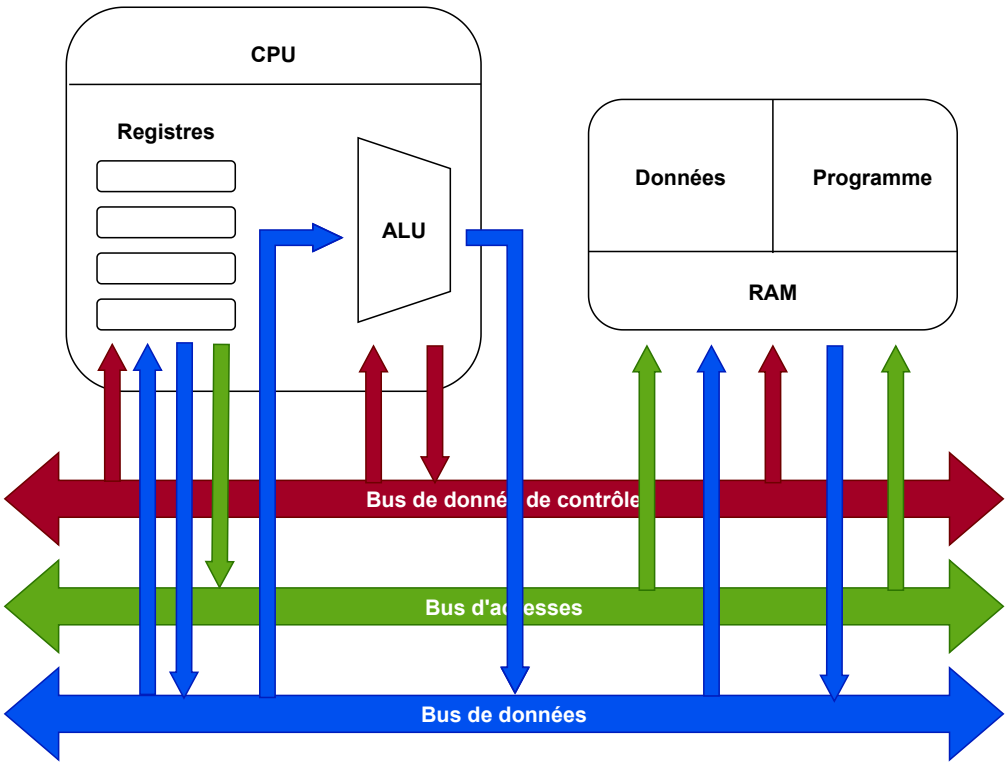

FIGURE 3 – Architecture d'un CPU

Registres

Les registres sont des parties du CPU capables de stocker des données. Leur fonctionnement est semblable à des mémoires RAM, la différence entre les deux est que les cellules mémoires des registres sont composés uniquement de portes logiques. La taille des données pourront être stockées dans ce types de mémoires est très faible comparées à des RAM, mais la vitesse d'écriture

et de lecture des registres est très élevée par rapport au autre support de stockage de données.

Il existe cinq types différents de registre dans le CPU qu'on cite ci-dessous :
- Accumulateur (AC) : stock les résultats de calcul ;
- Registre d'instruction (IR) : stock l'adresse mémoire de l'instruction à exécuter ;
- Registre d'adresse mémoire (MAR) : stock l'adresse mémoire de la donnée à traiter ;
- Registre de donnée mémoire (MDR) : stock la donnée qui est encours de traitement ;
- Pointeur d'instruction (PC) : stock l'adresse mémoire de l'instruction en cours d'exécution.

L'unité arithmétique logique

C'est le noyau du CPU et elle comporte de nombreux circuits logiques. Cette unité s'en sert de ces circuits afin d'exécuter différentes opérations. Parmi ces opérations on trouve des opérations arithmétiques comme l'addition, la soustraction ou encore la multiplication. L'ALU est également capable d'effectuer des opérations logiques comme la comparaison de nombres binaires afin de vérifier s'il sont identiques ou non.

L'unité de contrôle

C'est cette unité qui décode les instructions à exécuter et contrôle le fonctionnement des autres parties du CPU. Une fois une instruction est reçues par cette unité, qui est à la base qu'un ensemble de chiffres binaires, elle enverra des commandes à l'ALU pour lui indiquer de quelle opération s'agit-il. Cette unité contient également un horloge. Cette dernière contrôle la fréquence à laquelle les calculs sont effectués par le processeur.

Mémoire

Une mémoire RAM va servir à stocker les instructions que le processeur doit réaliser et également les données sur lequel il doit opérer. Deux registres par la suite sont nécessaires pour s'en servir de cette mémoire : un pour stocker l'adresse de la RAM dont on vient lire où écrire, et l'autre pour stocker la donnée manipulée lors de cette échange.

Les bus

Les composants que nous avons déjà cités sont tous reliés et cela à l'aide de ce qu'on appelle des bus de données. Il y'a des bus qui transport des données, d'autre des adresses et d'autres des instructions.

Avec cette dernière partie en clore alors les éléments de base dont dispose un processeur dit à un coeur de calcul. Cependant, le type de CPU le plus fréquent en informatique reste les processeurs à multi-coeur (voir figure.11). C'est des processeurs qui contiennent plusieurs coeur de calcul physique comme celui que nous avons décrit avant. Cela permet d'effectuer un nombre très important de calcul en parallèle et d'accélérer le temps d'exécution des programmes. Il existe aussi une API connue sous le nom de "OpenMP" qui nous permet d'exploiter cette architecture multi-coeurs et qui facilite la parallélisation des calculs .

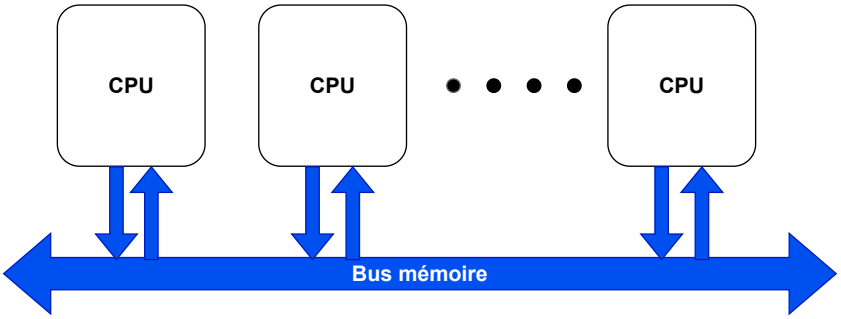

FIGURE 4 – Architecture d'un CPU multi-coeur

2.4 Architecture des GPU des cartes NVIDIA et CUDA :

L'unité de base dans un GPU est le SP (Scalar Processor) il s'agit d'une unité de calcul pour les nombres entiers et flottants. L'agrégation de plusieurs SP constitue un SM (Streaming Multiprocessor) et le GPU même est constitué de plusieurs SM. Ci-dessous un exemple simplifié des architectures NVIDIA classiques :

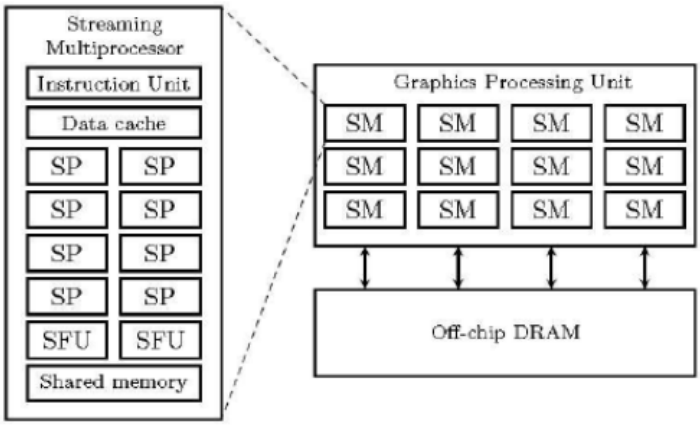

FIGURE 5 – Architecture NVIDIA classique

L'API CUDA (Compute Unified Device Architecture) développé par NVIDIA sert à l'exploitation des ressources des processeurs graphiques embarqués sur les cartes NVIDIA, afin de réaliser des calculs généraux à la place des processeurs centraux. Ceci nous permet de "offloader" les calculs parallélisables à la place d'être contraint par l'aspect séquentiel du fonctionnement des CPUs (comme la somme de deux tableaux). Ce principe est connu sous le nom GPGPU (general-purpose computing on graphics processing units).

L'unité de base dans CUDA est le thread, plusieurs threads constituent un bloc et plusieurs blocs constituent une grille qui exécute notre kernel CUDA. Pour faciliter la compréhension, on peut associer le grid au GPU complet et les blocs aux SMs, mais en vérité on peut avoir plusieurs blocs dans chaque SM selon le nombre de SMs disponibles dans le GPUs et le nombre de blocs dans la grille.

On peut donc pour un grid et selon le nombre de thread que l'on souhaite avoir déclaré le nombre de blocs que l'on veut et le nombre de threads dans chaque bloc. L'organisation des blocs dans le grid et l'organisation des threads dans chaque bloc est en 3D :

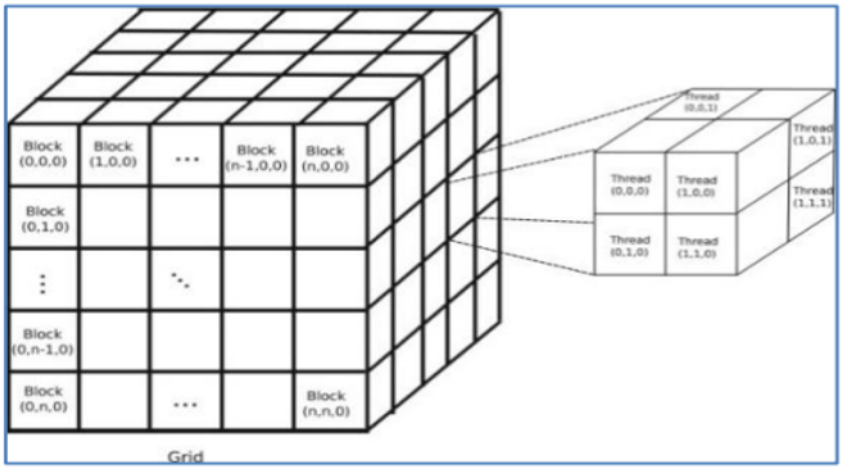

FIGURE 6 – Représentation d'une grille CUDA

On peut donc organiser nos blocs en 1D, 2D ou 3D et organiser les threads dans les blocs de la même manière. On peut également identifier les threads ou les blocs en cours d'exécution en utilisant les variables CUDA suivants :

- threadIdx.x, threadIdx.y, threadIdx.z : ID des threads
- blocDim.x, blocDim.y, blocDim.z : nombre de threads déclarés
- blocIdx.x, blocIdx.y, blocIdx.z : ID des blocs
- gridDim.x, gridDim.y, gridDim.z : nombre de blocs déclarés

Chaque thread possède ses propres registres et sa mémoire locale et il a accès à la mémoire globale du GPU. Et chaque bloc a une mémoire partagée entre les threads du bloc (shared memory).

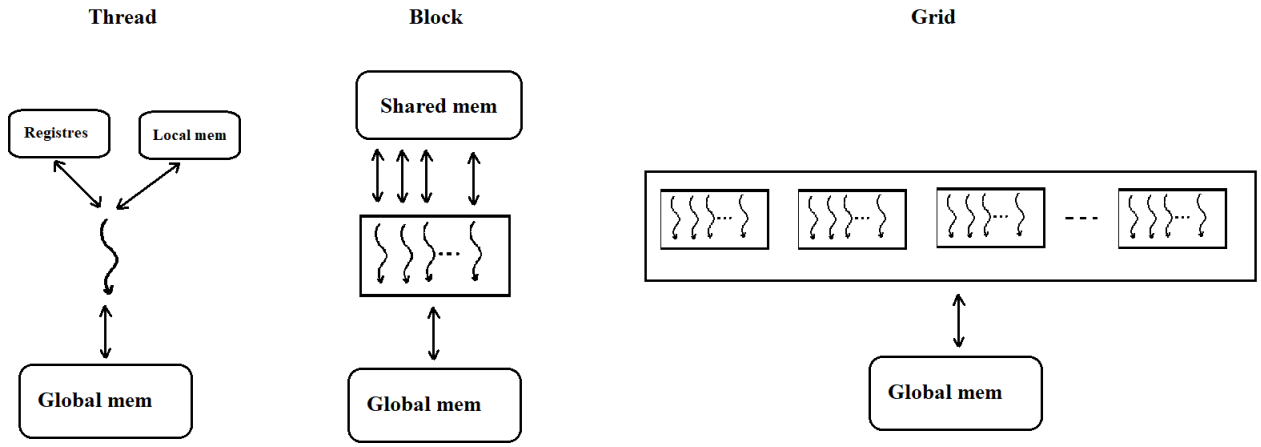

FIGURE 7 – Communication des threads, blocs, grille avec les mémoires

Afin de de compiler un projet utilisant CUDA, il faut ajouter le compilateur NVCC dans notre fichier make ou dans les dépendances de notre projet. Comme il faut bien suivre les étapes suivantes :

— Déclarer les données à traiter par le GPU par des pointeurs et faire une allocation dans la mémoire GPU en utilisant ces pointeurs et les tailles souhaitées.
— Copier les données (à traiter par le GPU) de la mémoire hôte (CPU) vers la mémoire du device (GPU), également appelée transfert host-to-device.
— Charger le programme GPU et exécutez (le programme est appelé kernel) : ici on lance le kernel qui est distingué par la directive `__global__`. Le corps de ce kernel est écrit dans un fichier sous l'extension `.cu`.
— Copier la mémoire du GPU vers la mémoire de l'hôte pour récupérer les résultats de calculs, ce transfert est également appelé transfert device-to-host.

3 Mandelbrot

3.1 Principe

L'ensemble de Mandelbrot tire ses origines de la dynamique complexe, un domaine défriché par les mathématiciens français Pierre Fatou et Gaston Julia au début du 20ème siècle. La première représentation de cet ensemble apparaît en 1978 dans un article. Le 1er mars 1980, Benoît Mandelbrot obtient pour la première fois, une visualisation par ordinateur de cet ensemble.

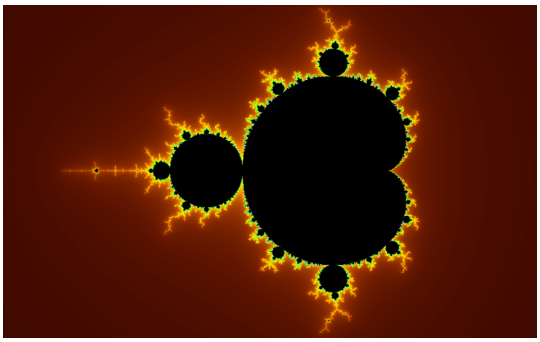

FIGURE 8 – La représentation de l'ensemble Mandelbrot

En fait, il s'agit d'un ensemble d'une immense complexité, il a une structure fractale (l'image globale se retrouve en plus petit dans certaines parties de l'ensemble). Cette structure est construite par une relation de récurrence, liant des nombres complexes entre eux. La suite de Mandelbrot se définit comme suit :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

FIGURE 9 – La suite Mandelbrot

Où :
Z est le nombre complexe sur lequel on applique la suite.
C est un autre nombre complexe quelconques.

En pratique, pour obtenir les figures de Mandelbrot, il faut parcourir tout le plan complexe, et pour chaque point C de coordonnées (x,y) qui devient notre nombre complexe, on calcule

les valeurs de la suite un nombre d'itération arbitraire de fois. Ainsi, il faut fixer un seuil bien déterminé pour juger si le résultat est contenu dans un intervalle fini quel que soit le nombre de fois qu'on itérera la suite. Si c'est le cas, on peut dire que ce point appartient l'ensemble de Mandelbrot, et on colorie ce pixel en noir. Dans le cas contraire, nous affectons à ce pixel une couleur qui correspondent à la vitesse avec laquelle cette suite converge vers l'infini. Cette vitesse est interprété par le nombre d'itération au bout de laquelle on se trouve à l'infini (généralement une limite arbitraire fixée, par exemple 255). Afin de donner à cet ensemble l'aspect d'un fractale, nous devons calculer les nouveaux coordonnées (x,y) du point C, en fonction de ces coordonnées initiales, largeur et langueur de l'image, ainsi que une valeur qui caractérise le zoom effectué.

3.2 Code de base

Afin de pouvoir afficher des fractales en suivant le principe cité juste avant, nous avons réalisé un programme qui tourne sur CPU en se basant sur un code qui nous a été fourni. Nous avons développer une seule classe pour cela, nommée `Mandelbrot`, dont les méthodes et les attribues seront expliqués par la suite. Nous avons également eu recours à l'API "SFML" qui fournit une interface simple aux différents composants d'un ordinateur et facilite le développement d'applications multimédias.

Nous détaillerons dans ce qui suit le déroulement de ce programme.

Déroulement du code

On commence tout d'abord par choisir la taille des images à générer en spécifiant leur largeur et hauteur. On fixe également le nombre d'itérations maximal pour le calcul des suites de Mandelbrot. Une fois ces paramètres choisis, on crée un objet de la classe `Mandelbrot`. D'autre variables seront initialisées par défaut et on parle ici de `offsetX` et `offsetY` qui vont permet de ce déplacer sur l'image, et de `zoom` qui va permettre de zoomer.

A l'aide de l'API `SFML`, on crée une fenêtre sur l'écran qui va nous servir pour afficher les images de fractales et une variable `image` qui va contenir les couleurs des pixels de l'image à afficher. Cette API nous permettra par la suite également gérer les évènements qui se passent sur l'ordinateur. Les évènement qui vont nous intéresser son les suivant :

- Mouvement de la roulette de la souris ;
- Clique sur un bouton de la souris ;
- Clique sur un bouton du clavier.

Une fois cette fenêtre créé, nous allons traité les évènements cités avant. En fonction de ces derniers, nous actualisons l'image à afficher.

Nous allons maintenant détailler la façon dont laquelle on rafraîchie l'image à chaque fois. On commence par faire appelle à la méthode `CpuUpdateImage()` de la classe `Mandelbrot` qui assure l'actualisation de l'image toute en assurant l'aspect fractale de celle-ci. Cette méthode fait appel à la méthode `Cpuprocess()` qui se charge par le calcul de la suite de Mandelbrot, expliquée dans la section précédente. Cette fonction va prendre comme paramètre, les variables `offsetX`, `offsetY`, `zoom`, la taille de l'image et un tableau où on va stocker par la suite les résultats de calcul. La fonction `CpuUpdateImage()` va déterminer pour chaque pixel si la suite de Mandelbrot diverge ou pas et va mettre dans le tableau de résultats le nombre d'itérations effectués.

Ce tableau par la suite est exploité par la méthode `Colorize()` pour associer à chaque nombre d'itérations de chaque pixel une couleur, puis stocker cette informations dans la variable `image`.

On affiche par la suite le contenue de la variable `image` sur l'écran à l'aide de l'API "SFML". La figure suivante montre l'image generée au début du programme :

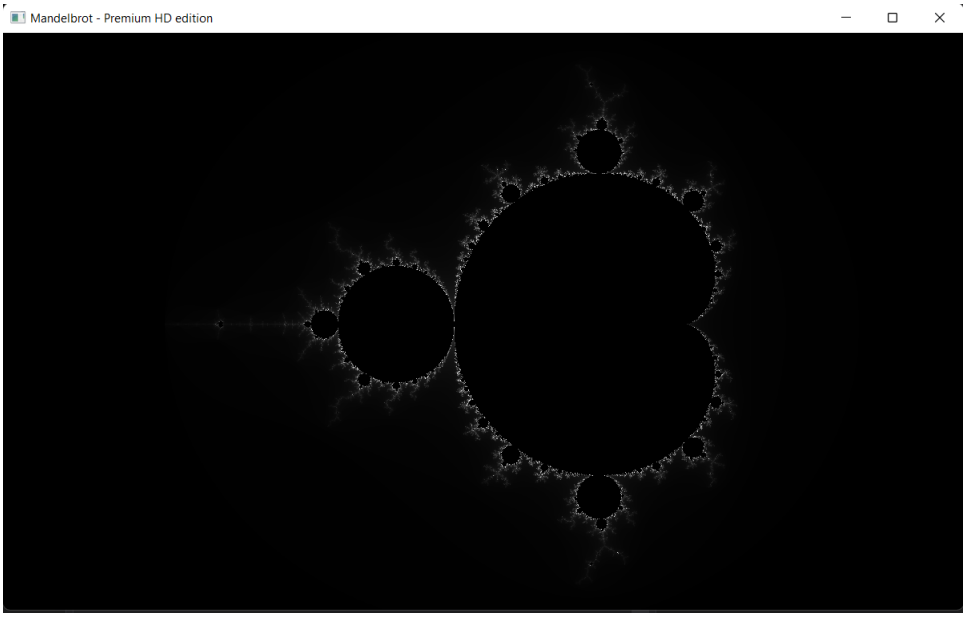

FIGURE 10 – Fractale de Mandelbrot générée par le programme

On remarque par la suite que ces calculs sont assez nombreux et en conséquence le rafraîchissement des images prend beaucoup de temps. D'où la nécessité de recourir à des optimisations.

3.3 Parallélisation de calcul avec OpenMP

La première optimisation qui a été mise est oeuvre est l'utilisation de l'api "OpenMP".

OpenMP implémente le multithreading, une solution de parallélisation pour la quel un thread principal (une série d'instruction consécutives) se divise en des sous-threads et le système divise la tache entre eux. Les sous-threads s'exécute ensuite de façon simultanée.

La section du code à paralléliser est indiqué par une directive du compilateur qui entraînera la formation des threads avant l'exécution de celle-ci. Une fois cette section exécutée, les threads crées précédemment se rejoignent dans le thread principale. La figure ci-dessous contient un exemple qui illustre ce principe :

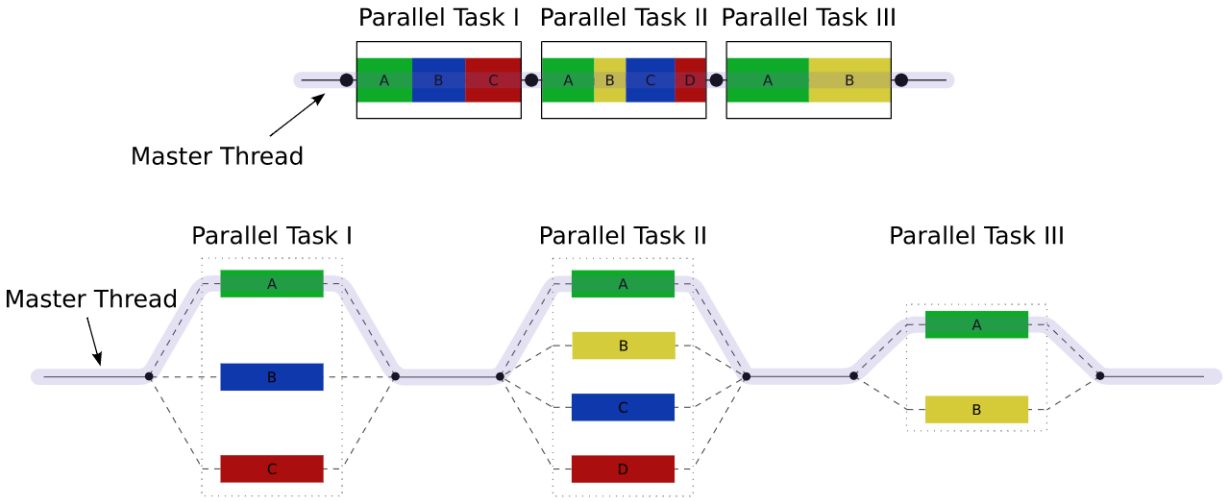

FIGURE 11 – Le principe du multithreading

Dans cette exemple, trois sections du code sont à paralléliser (Parallel Task I, Parallel Task II, Parallel Task III). Le thread principal (Master Thead) se divise en trois thread au départ pour réaliser la première tache. Une fois cette tache réaliser, les sous-threads crées se rejoignent dans le thread principale et puis d'autre sous threads sont crées. On continue ainsi par créer puis recombiner les différents threads jusqu'à réaliser toutes les sections à paralléliser.

Voila que le principe de multithreading est claire, on l'exploitera maintenant dans notre code. Pour cela nous avons chercher la partie du code qui nécessite le plus de calcul et qui alourdie la vitesse d'exécution de notre code. Cette partie concerne bien évidement les méthodes du rafraîchissement de l'image ainsi que la fonction qui calcul la suite de Mandelbrot, on parle alors de la boucles suivante du programme :

```
1 for (int y = 0 ; y < IMAGE_HEIGHT ; y++) {
2     for (int x = 0 ; x < IMAGE_WIDTH ; x++) {
3
4         ...
5
6         }
7 }
```

C'est ici où on balaye toute les pixels de notre image pour effectuer les calculs nécessaire à déterminer les couleurs adéquates pour chacun d'entre eux.

Afin de paralléliser cette section avec OpenMP, il a suffi d'ajouter une directive de compilation de la façon suivante :

```
1 #pragma omp parallel for num_threads(std::thread::hardware_concurrency())
      schedule(dynamic)
2 for (int y = 0 ; y < IMAGE_HEIGHT ; y++) {
3     for (int x = 0 ; x < IMAGE_WIDTH ; x++) {
4
5         ...
6
7         }
8 }
```

!

où :

— `omp parallel for` : indique que la prochaine boucle "for" est à paralléliser ;
— `num_threads()` : fixe le nombre maximale de threads à utiliser ;
— `std::thread::hardware_concurrency()` : renvoie le nombre de coeurs (physiques et logiques) du CPU utilisé ;
— `schedule(dynamic)` : permet de répartir les calculs dynamiquement pour qu'un coeur du CPU prenne en charge un thread dès qu'il est libre.

3.4 Parallélisation de calcul avec CUDA :

Après avoir partagé les calculs lourds sur l'architecture multi-cœurs du processeur, dans cette phase nous désirons d'associer ces calculs aux ressources GPU en se basant sur les informations évoquées précédemment toute en respectant les étapes énoncées à la section (2.5 Architecture des GPU des cartes NVIDIA et CUDA).

Alors nous commençons par la configuration du périphérique GPU en utilisant la fonction cudaSetDevice(). Puis, nous déclarons un pointeur `gpu_ptr` qui va pointer vers le tableau dans le quel nous allons stocker les résultats de calcul de la suite de Mandelbrot pour chaque pixel. Par la suite, nous allouons ce tableau dans la mémoire globale du GPU en utilisant son pointeur toute en précisant sa taille en utilisant la fonction `cudaMalloc()` de la bibliothèque CUDA.

```
cudaMalloc((void**)gpu_ptr, f_width * f_height * sizeof(float));
```

Toutes ces étapes sont réalisées dans la méthode de construction des objets Mandelbrot.

Puisque dans notre cas nous n'avons pas des données existant sur la mémoire globale du CPU qui vont être traitées par le GPU, nous n'aurons pas besoin de faire des transferts de type hosttodevice et par conséquent nous passons aux étapes qui consistent au chargement du programme GPU en appelant la fonction de lancement du kernel `comput_cuda_s()` et au transfert device-to-host pour transférer le tableau des résultats existant sur la mémoire sur le GPU de la carte NVIDIA vers la mémoire de CPU en utilisant la fonction :

```
1 cudaMemcpy (ptr, gpu_ptr,IMAGE_HEIGHT * IMAGE_WIDTH * sizeof(float),
    cudaMemcpyDeviceToHost);
```

Ces deux étape seront effectuées dans une méthode `GpuUpdateImage()` de la classe Mandelbrot.

Dans ce qui suit nous détaillons la partie de l'exécution de kernel.

En fait la fonction `compute_cuda_s()` appelée par la méthode `GpuUpdateImage()` et déclarée dans le fichier kernel d'extension `.cu`, consiste à dimensionner la grille et les blocs en 2D, à calculer la dimension de chaque bloc sur l'axe X et Y par l'intermédiaire de deux formules assurant que le nombre de threads défini est supérieur au nombre de pixel et de lancer le kernel en partageant le calcul sur les ressources prédéfinie de façon à associer à chaque thread un pixel.

Le kernel qui est repéré par la directive __global__ est vu comme le code à exécuter par chaque thread d'une manière séquentielle. Il commence par l'attribution à chaque thread un ID bidimensionnel en utilisant les variables CUDA (c'est-à-dire chaque thread eq pixel, identifié par un cordonné (X,Y)), par la suite, il précise aux threads concourant les calculs à faire. Puisque le rafraîchissement de l'image désormais à réaliser par notre kernel fait appel à une méthode `Cpuprocess()`, il faut redéfinir cette fonction dans le fichier `kernel.cu` en utilisant la directive __device__ sous un autre intitulé `process_cuda_s()`. Après la réalisation de ces modifications il nous reste que de bien gérer les dépendances notre projet et de bien paramétrer le compilateur NVCC.

3.5 Exécution et résultats :

3.5.1 Benchmark Mandelbrot :

Le benchmark pour le projet de Mandelbrot, consiste à mettre à jour l'image un certain nombre de fois (nTests) avec des paramètres constants, tout en affichant le délai moyen consommé pour une seule actualisation.
Cette mesure est effectuée en récupérant l'heure actuelle avant et après l'exécution de ce banc d'essai, en utilisant la méthode `now()` de la classe `steady_clock` de la bibliothèque standard chrono. Par la suite, nous effectuons la différence, on la divise par le nombre de tests et on l'affiche sur la console en microsecondes.
Cette valeur va être mesuré pour une exécution sur le CPU avec et sans utilisation de l'API OpenMP, puis sur le GPU de NVIDIA. Ces tests vont être également réalisés sur deux machines différentes.

3.5.2 Caractéristiques des deux machines :

CPU et Système d'exploitation	Machine 1	Machine 2
Référence	Intel Core(TM) i5-10210U	Intel Core(TM) i5-10300H
Fréquence de base CPU	2.11 GHz	2.50 GHz
fréquence turbo max	4.2GHz	4.5GHz
Nbr de cœurs	4 cœurs physique, 8 logiques	4 cœurs physique, 8 logiques
Mémoires caches	L1=256Ko,L2=1Mo,L3=6Mo	L1=256Ko,L2=1Mo,L3=8Mo
Mémoire RAM	12 Go	16 Go
Architecture	x86	x64
Système d'exploitation	Windows 10 Professionnel	Windows 11 Famille

Carte NVIDIA	Machine 1	Machine 2
Référence	NVIDIA GeForce MX130	NVIDIA GeForce RTX2070 (mobile)
Fréquence GPU	1122 MHz	1440 MHz
Nbr de cœurs CUDA	384	2304
Mémoire VRAM	4096 Mo	8192 Mo
Architecture	Maxwell	Turing
API supportés	DirectX : 12 (11_0), Modèle de shader : 5.1, OpenGL : 4.6, Vulkan : 1.1.126, OpenCL :1.2 , CUDA : +	DirectX : 12 Ultimate (12_1) Modèle de shader : 6.5, OpenGL : 4.6, Vulkan : 1.2.131, OpenCL :1.2 , CUDA : +
Capacité de calcul	5.0	7.5

L'architecture des GPU utilisés n'étant pas la même, on donnera quelque détails sur ce sujet.
Nous allons commencer par l'architecture Maxwell. Celle-ci se caractérise par les SMMs (Streaming Maxwell Multiprocessors), dont chacun englobe 128 cœurs CUDA. Ces derniers, sont divisés sur 4 sous-ensembles de chaque SMM. On peut dire alors que 4 "cœurs majeurs" sont séparés dans un SMM, et deux caches L1 chacun est partagé par deux cœurs majeurs avec 4 unités de texture. Cette division présente l'avantage de cette architecture, en fait, le warp scheduler n'a le contrôle que de son propre « cœur majeur » et rien n'est partagé entre les 4 cœurs principaux, à l'exception des unités FP64 et Texture (par les ordonnanceurs warp) rien n'est partagé entre les 4 cœurs principaux, à l'exception des unités FP64 et Texture (par warp schedulers). La représentation de cette architecture sera figurée à l'annexe.
Passant maintenant à la deuxième architecture qui est la Turing. Celle-ci à introduit une nouvelle architecture de processeur, le Turing SM (Streaming Multiprocessors), qui améliore considérablement efficacité d'ombrage, atteignant une amélioration de 50% des performances fournies par cœur CUDA par rapport à la génération Pascal. Ces améliorations sont rendues possibles par deux changements architecturaux clés. Tout d'abord, le Turing SM ajoute un nouveau chemin de données entières indépendant qui peut exécuter des instructions en même temps que le chemin de données mathématiques à virgule flottante. Turing à également introduit le ray tracing [1] en temps réel qui permet à un seul GPU de restituer des jeux 3D visuellement réaliste et modèles professionnels complexes avec des ombres, des reflets et des réfractions physiquement précis. Les nouveaux coeurs RT de Turing accélèrent le lancer de rayons et sont exploités par les systèmes et des interfaces telles que la technologie de traçage de rayons RTX de NVIDIA. La représentation de cette architecture sera incluse en annexe.

3.5.3 Résultats

Pour être plus objectif dans notre comparaison nous avons défini un protocole de test qui consiste a réaliser le Benchmarck, où l'image sera actualisée 10 fois à chaque exécution, juste après le démarrage des deux ordinateurs pour assurer que les conditions initiales des deux machine son presque les même.

Après avoir configurer le projet sur les deux machines, nous avons lancé notre Benchmark. Les résultats de ces tests sont représentés sur le graphe suivant :

1. Technique de calcul d'optique qui consiste à simuler le parcours inverse de la lumière.

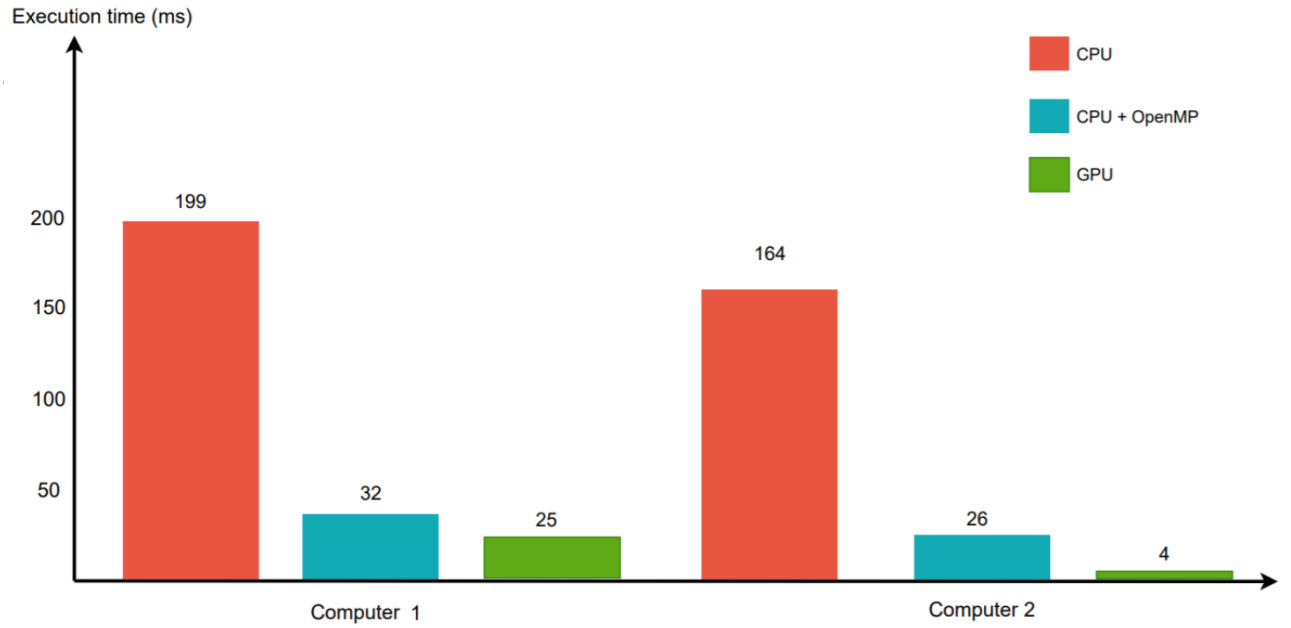

FIGURE 12 – Résultat de l'exécution du Benchmark Mandelbrot

Pour la partie CPU, on voie que comme se qui était attendu, l'utilisation de l'API OpenMP à améliorer avec un facteur de presque 6 les performances de calcul. On s'attendait à avoir un résultat pareil du fait que le rafraîchissement de l'image se fait sur plusieurs threads ai lieu d'un seul avec OpenMP. Ci on compare la performances des deux CPU on verra que celui de la machine 2 à dépasser celui de la machine 1. Cela revient au fait que les caractéristiques techniques du CPU d'Intel i5 10300H dépasse celle du i5 10210U. Le premier dispose d'une fréquence de calcul plus importante allant jusqu'à 4.5 Ghz contre 4.2 Ghz pour le deuxième CPU. En plus la vitesse de la mémoire RAM de la deuxième machine dépasse celle de la première (2933 Mhz contre Mhz). Ce dernier facteur est assez important tenant compte de la nature de notre programme où les accès mémoires sont accès fréquent.

Lorsqu'on bascule sur le codé GPU, on voie que les performances en matières de temps de calcul s'améliore encore plus. Ce qui est logique vu que les processeurs graphiques dispose de plus de coeurs de calcul que les CPU. Pour la carte graphique MX130 de Nvidea, il a presque pas eu d'amélioration par rapport à l'utilisation de l'api OpenMP. Ceci est probablement du à un mauvais réglage sur la machine 1. Le fait que la machine 2 dépasse la machine 1 dans ce test est également raisonnable. La carte graphique RTX2070 est la plus performantes et avec une très grande marge. Elle dispose de 2304 coeurs de calcul avec une fréquence de 1440 Mhz contre 384 coeurs avec une fréquence de 1122 MHz pour la MX130. Également la mémoire vidéo RAM de la RTX2070 est de 8 Go contre 4 Go pour la MX130.

4 Raytracer

4.1 Principe

Le Ray Tracing est une méthode de calcul d'optique utilisée en informatique et spécifiquement dans les jeux videos. En se basant sur une méthode de rendu par illumination, elle reproduit le parcours inverse de la lumière en calculant les éclairages qui vont de la caméra

vers les objets puis vers les sources lumineuses. Cette technique est basée sur les principes de réflexion et de réfraction.
Cela consiste à émettre un premier rayon lumineux depuis la caméra vers chaque pixel de l'image. Le rayon est dévié par un objet et un point d'impact est identifié. Depuis ce dernier, de nouveaux rayons sont alors simulés, en direction de chaque source lumineuse.

En d'autres termes, Le Ray Tracing inverse la logique. Plutôt que de se baser sur une source lumineuse dans une scène, la caméra du joueur projette des rayons qui, au contact d'un objet, vont permettre de calculer sa réflexion, sa réfraction.

Ce qui justifie notre choix pour ce projet c'est la complexité de ce dernier et le fait qu'il se base sur beaucoup de calculs qui vont saturer la capacité d'un ordinateur, cela s'accorde bien avec le thème de notre projet de cette année.

4.2 Explication du code

4.2.1 Création de l'image

Pour pouvoir visualiser cet effet, on commence d'abord par faire l'affichage de l'image, alors pour le faire il existe plusieurs méthodes parmi eux, l'utilisation de l'API "SFML" comme dans le projet précédent, mais dans celui là on a choisit la méthode la plus simple pour afficher une image, c'est d'écrire toute les pixels dans un fichier ppm qui va à son tour traduire les pixels en une image.

On affiche une image simple avec des pixels aléatoire et on obtient ce résultat :

FIGURE 13 – Image Ppm

4.2.2 Création des rayons

Pour créer la forme de nos objets qui vont être que des sphères dans ce projet parcequ'ils sont les plus simple à manipuler , on va envoyer des rayons depuis la caméra afin de déterminer les objets qui vont s'intersecter avec, puis on calcule la couleur pour chaque point d'intersection.

On représente un rayon par la simple fonction mathématique $\mathbf{P(t) = A + tB}$
P étant une position 3D au long d'une ligne en 3D, **A** est l'origine du rayon et **B** est la direction. Alors pour chaque t différent on aura un P(t) qui se déplace au long du rayon

FIGURE 14 – Représentation d'un rayon

4.2.3 Création des sphères

Étant donné les coordonnées d'une sphère , on doit chercher les points dans nos rayons émis par la caméra qui vont intersecter avec notre sphère pour qu'on puisse colorer ces points, et finalement aboutir à une forme de disque sur notre image.
On sait que l'équation d'une sphère centrée sur l'origine et de rayon R est :

$$x^2 + y^2 + z^2 = R^2$$

En d'autres termes, si un point donné (x,y,z) se trouve sur la sphère, alors :

$$x^2 + y^2 + z^2 = R^2$$

Si le point donné (x,y,z) est à l'intérieur de la sphère, alors :

$$x^2 + y^2 + z^2 < R^2$$

et si un point donné (x,y,z) est à l'extérieur de la sphère, alors :

$$x^2 + y^2 + z^2 > R^2$$

Cela devient plus compliqué si l'origine de la sphère est en (C_x, C_y, C_z) :

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = R^2$$

Pour simplifier pour la suite on représente les coordonnées par des vecteurs, alors le vecteur entre le le Point $P = (x, y, z)$ qui existe sur le rayon et le centre de la sphère $C = (C_x, C_y, C_z)$ est $(P - C)$, Donc : Alors $(P - C)$

$$(P - C).(P - C) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2$$

Ce qui donne :

$$(P - C).(P - C) = R^2$$

On peut remarque alors que tout point P qui satisfait à cette équation est sur la sphère. On veut savoir si notre rayon $P(t) = A + tb$ touche la sphère à un endroit quelconque. S'il touche la sphère, il existe un certain t pour lequel P(t) satisfait l'équation de la sphère. On recherche donc tout t pour lequel cela est vrai :

$$(P(t) - C)(P(t) - C) = R^2$$

En remplaçant P(t) par $At + b$ on trouve :

$$t^2 b.b + 2tb(A - C) + (A - C)(A - C) - R^2 = 0$$

Les vecteurs et r dans cette équation sont tous constants et connus. L'inconnue est t, et l'équation est une quadratique.
On peut résoudre t et il y a une partie racine carrée qui est soit positive (ce qui signifie deux solutions réelles), soit négative (ce qui signifie aucune solution réelle), soit nulle (ce qui signifie une solution réelle). En graphisme, on a :

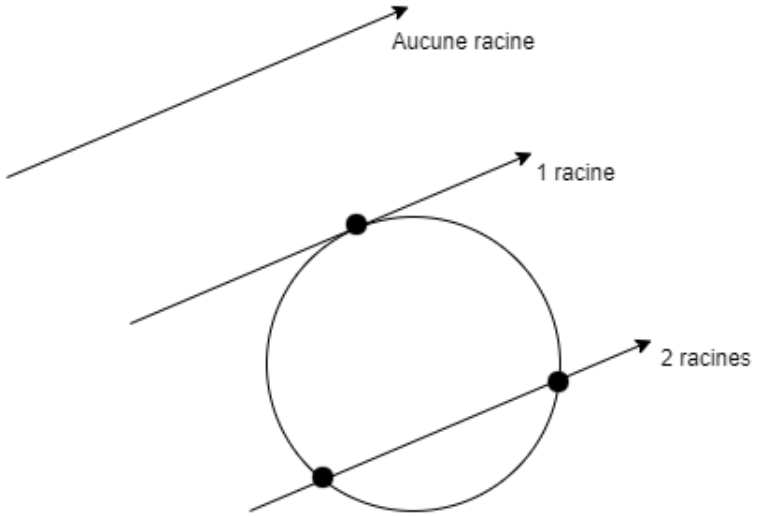

FIGURE 15 – Intersections rayons - sphère

Si nous intégrons ce calcul dans notre programme, nous pouvons le tester en colorant en rouge tout pixel qui touche une sphère qu'on place à -1 sur l'axe z :

FIGURE 16 – Première sphère

Il manque encore quelques étapes avant la fin de cette application comme l'ombrage, la réflection des rayons et d'autres sphères pour visualiser l'effet raytracing.

4.2.4 Ajout de l'aspect ombre

Pour pouvoir faire l'ombre d'une sphère, il faut d'abord trouver une normale à la surface. Il s'agit d'un vecteur perpendiculaire à la surface au point d'intersection. Pour une sphère, la normale extérieure est dans la direction du point d'impact moins le centre :

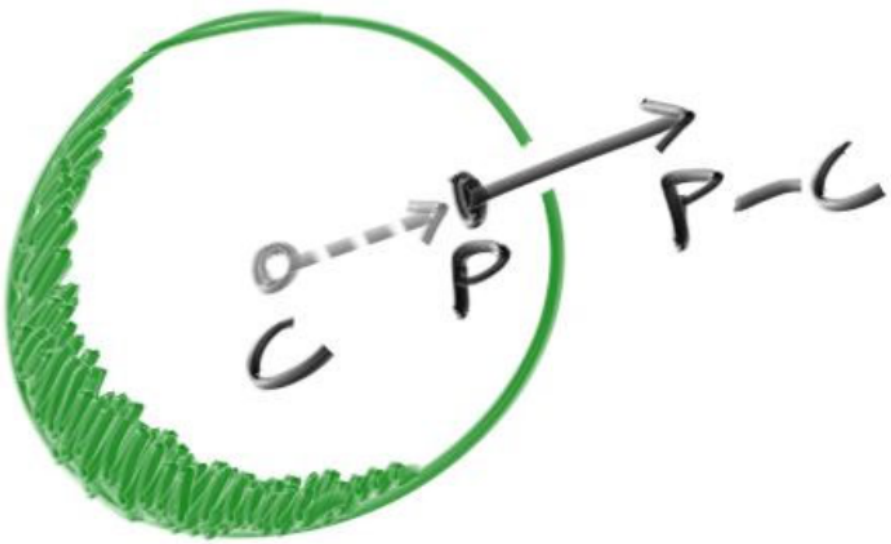

FIGURE 17 – la normale d'une sphère

Une fois on a la normale de la sphère il suffit alors de faire varier la couleur des pixels qui intersectent avec la sphère en fonction de la distance de ces derniers avec la normale. Après avoir ajouté ceci dans le code, on trouve le résultat suivant :

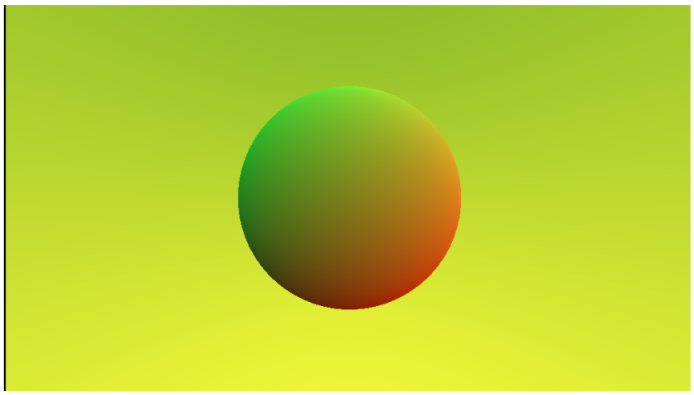

FIGURE 18 – Effet d'ombre

Comme prévu le côté où on a choisit la normale est plus claire que le côté opposé où on peut clairement voir qu'il est plus sombre.

4.2.5 Diffusion des matériaux

Afin de créer l'effet Raytracing, on va implémenter l'effet de la réflexion des rayons.
On commence d'abord par la création des sphère de matière métal, qui va non seulement absorber une partie des rayons qui sont émis vers elle, mais refléter une partie également.
La direction de la lumière qui se reflète sur une surface est aléatoire, alors 2 rayons envoyé de

la même position peuvent avoir des trajectoires différentes après la réflexion comme illustré sur la figure ci-dessous.

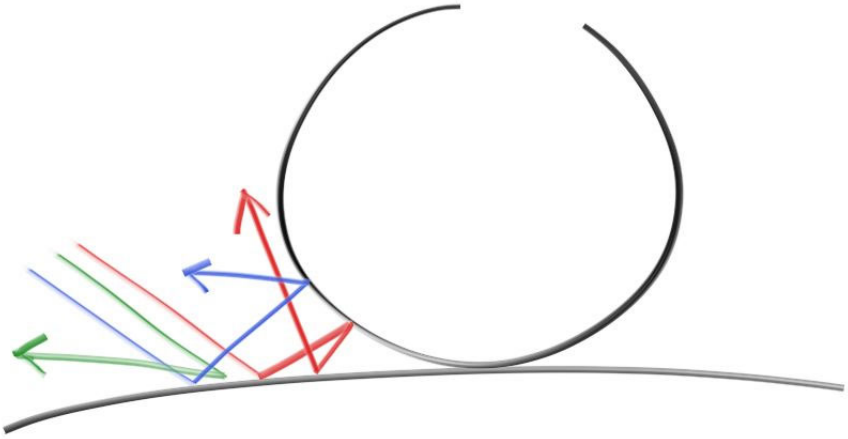

FIGURE 19 – Réflexion des rayons

Après avoir implémenté cet effet dans notre code, on crée 2 sphère de ce type de matériel et une sphère normale pour visualiser les réflexions, et on obtient le résultat suivant :

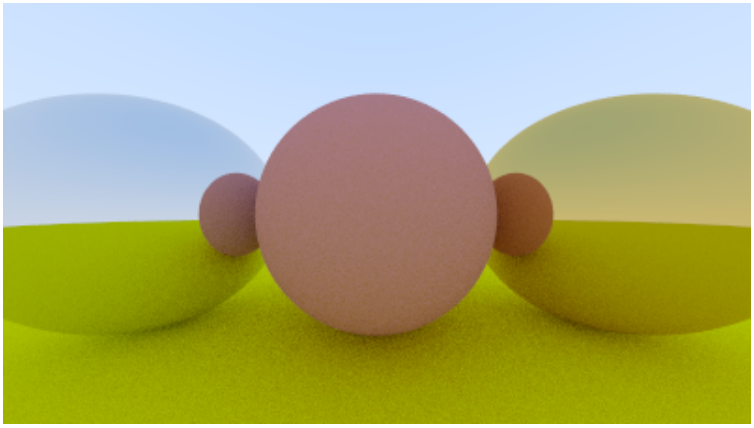

FIGURE 20 – Effet Réflexion

4.2.6 Final look

Maintenant, nous avons composé tous les éléments nécessaire pour la création de l'effet ray tracer, on crée alors plusieurs sphères de couleurs et de positions aléatoires et on obtient le résultat suivant :

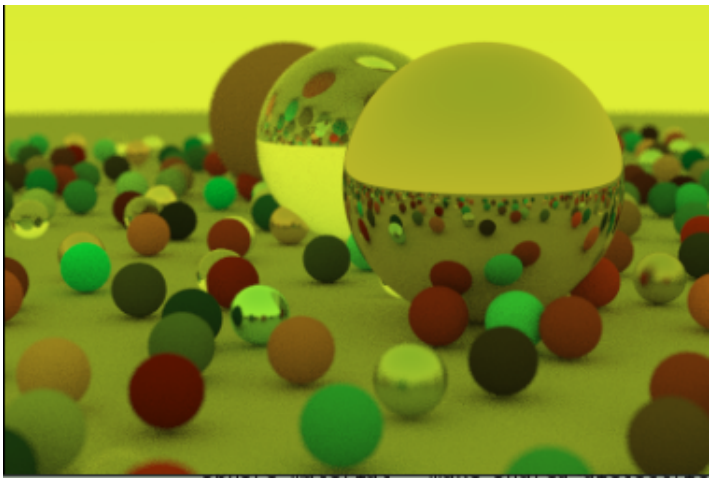

FIGURE 21 – Résultat final

4.3 Parallélisation de calcul avec OpenMP

Puisque la manière d'affichage de l'image était de tout simplement calculer chaque pixel et l'écrire dans un fichier, la méthode de parallélisation directe ne marche pas comme on le veut. En effet puisque l'ordre de l'écriture des fichiers est important pour l'affichage, si on associe alors la tâche aux plusieurs threads qui vont écrire dans le fichier les pixels qu'ils ont calculé, on préservera pas l'ordre des pixels, puisque les threads distribue les pixels à calculer entre eux et écrivent en même temps dans le fichier.
Alors pour résoudre ce problème, on va créer un buffer on va stoquer les pixels que les threads calculent afin de les écrire en ordre sans l'utilisation des threads.
Le coût d'utilisation d'un buffer est négligeable devant l'optimisation fourni par la parallélisation du calcul.

4.4 Parallélisation de calcul avec CUDA

Par manque de temps, nous n'avons pas eu la possibilité d'exploiter cette optimisation et de réaliser les calculs sur GPU.

4.5 Résultat

Après avoir testé les deux approches sur les deux ordinateurs, on trouve les résultats suivant :

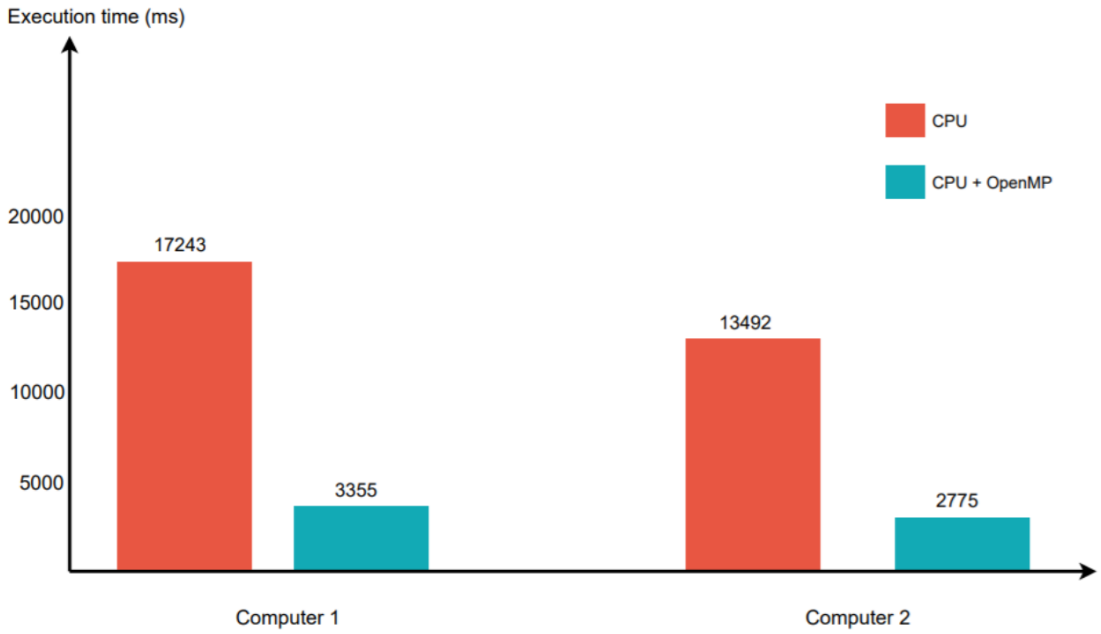

FIGURE 22 – Effet d'ombre

Comme prévu, la parallélisation des calculs donne des résultats beaucoup plus supérieur que sans l'utilisation des threads.

5 Conclusion générale

Le premier objectif de ce projet était de générer un ensemble de Mandelbrot et une image basée sur le principe de rayTracing toute en parallélisant les calculs en bénéficiant l'architecture multicœurs du CPU et des ressources GPU de la carte NVIDIA.

Concernant, le Mandelbrot nous avons réalisé notre objectif, nous avons comparé les performances de l'application en l'exécutant sur deux différentes machines et nous avons eu des résultats logiques et bien justifiés.

Cependant, durant notre étude nous avons confronté des contraintes techniques et sous contraintes de temps, nous ne sommes pas arrivés à optimiser le calcul du rayTracer sur GPU.
Alors, si nous avons désiré au préalable de réaliser entièrement notre projet c'est que effectivement dans notre perspective et si les conditions se présentent, nous aurons l'intention ardente de continuer sur ce même chantier pour paralléliser notre deuxième application en exploitant les ressources de la carte NVIDIA.

Finalement, nous admettons que cette expérience nous a permis enrichir nos connaissances préliminaires en programmation, en architecture numérique avancée et en microinformatique.
Au-delà de l'aspect technique, ce projet nous a présenté une formidable expérience, qui nous a permis aussi de perfectionner certaines compétences personnelles à savoir le travail collaboratif et la gestion des efforts.

6 Annexes

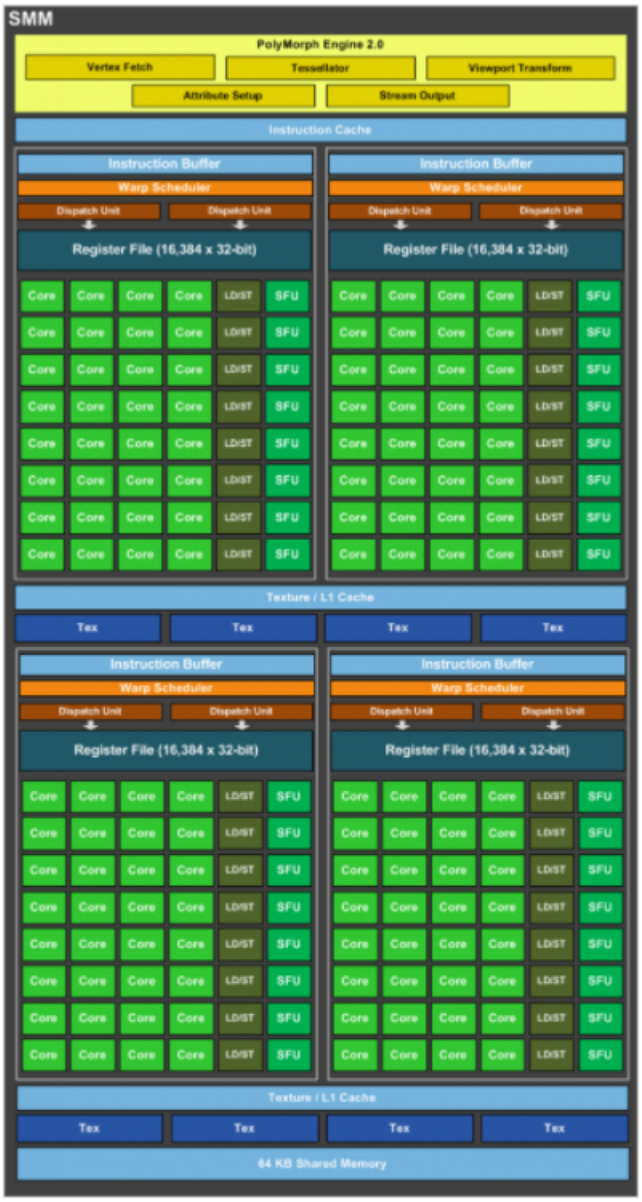

FIGURE 23 – L'architecture Maxwell de Nvidia

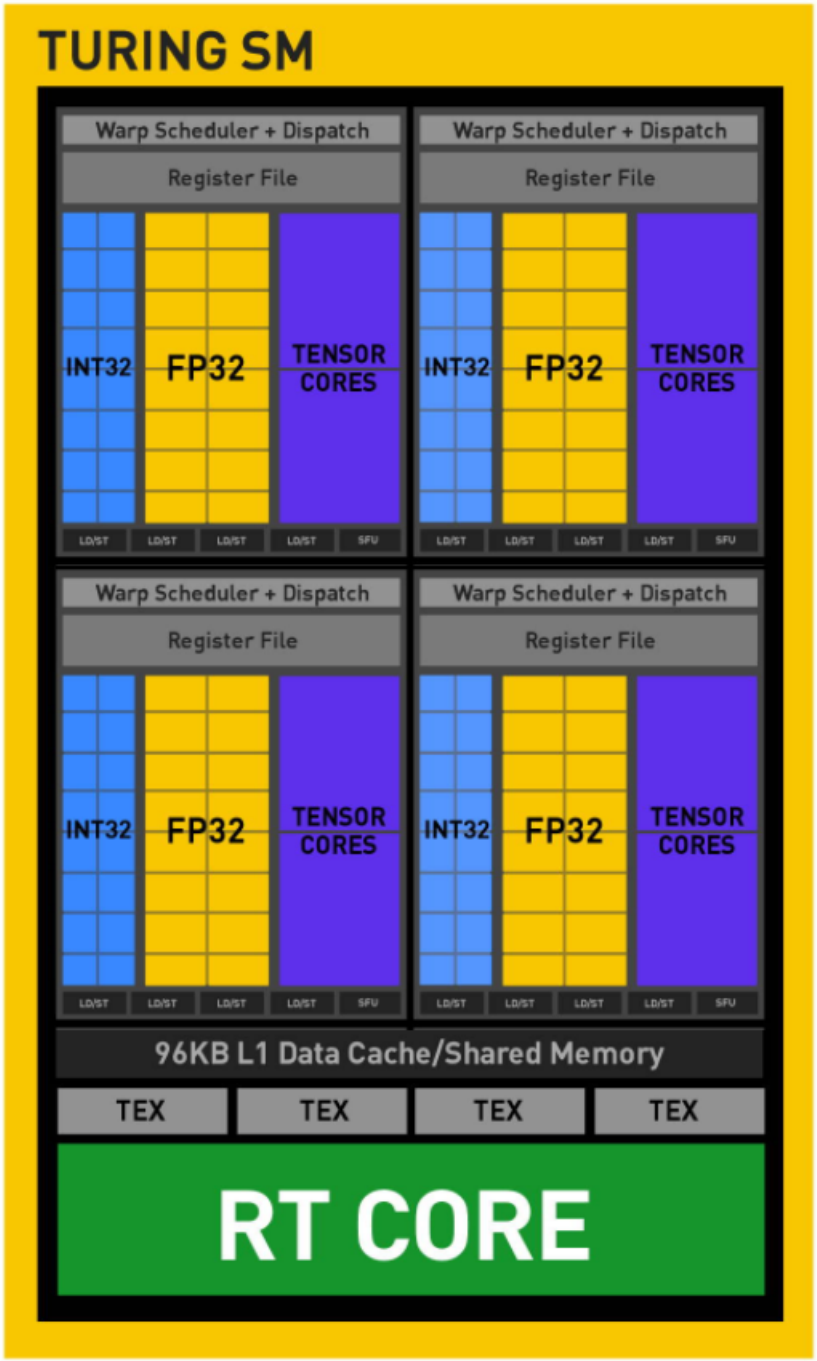

FIGURE 24 – L'architecture Turing de Nvidia