

République Algérienne Démocratique et Populaire  
 الجمهورية الجزائرية الديمقراطية الشعبية  
 Ministère de l'Enseignement Supérieur et de la Recherche Scientifique  
 وزارة التعليم العالي والبحث العلمي

---



المدرسة الوطنية العليا للإعلام الآلي  
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)  
École nationale Supérieure d'Informatique ex. INI  
(Institut National de formation en Informatique)

## Rapport de Travail Pratique (TP)

### Module : HPC - High Performance Computing

2<sup>ème</sup> année Cycle Supérieur (2CS)

Option : Systèmes Intelligents et Données (SID)

Groupe : 2SD1

---

### Thème :

Programmation parallèle avec CUDA  
Application à l'ensemble de Mandelbrot

---

Réalisé par :

- ABOUD Ibrahim
- BOUYAKOUB Rayane

Proposé par :

- M<sup>me</sup> HAICHOUR Amina Selma

---

Année universitaire : 2024-2025

## Table des matières

Table des matières .....	I
Liste des figures .....	II
Liste des tableaux .....	II
1. Introduction .....	1
2. Objectifs.....	1
3. Rappel du problème traité.....	2
4. Intérêt de la parallélisation .....	5
5. Caractéristiques du GPU .....	5
6. Stratégie de la parallélisation.....	6
7. Solution parallèle.....	7
7. Résultats et interprétations .....	11
8. Conclusion .....	13

## Liste des figures

Figure 1: Représentation visuelle de l'ensemble de Mandelbrot.	2
Figure 2: Impact du nombre d'itérations sur la précision de la représentation lors d'un zoom profond sur une bordure de l'ensemble de Mandelbrot.	3
Figure 3: Organisation des blocs en une grille de taille $M \times N$ .	9
Figure 4: Organisation des threads dans un bloc $n \times n$ .	10
Figure 5: Schéma récapitulatif de la solution parallèle.	11

## Liste des tableaux

Tableau 1: Tableau résumant les caractéristiques de l'architecture du GPU dans laquelle nous avons effectué la parallélisation.	5
Tableau 2: Tableau récapitulatif des mesures de performance entre la programmation parallèle Pthreads et CUDA.	12

## 1. Introduction

Les GPU (Graphics Processing Units) sont devenus un élément central du calcul haute performance, bien au-delà de leur rôle initial dans le rendu graphique. Leur architecture massivement parallèle les rend particulièrement adaptés à des tâches nécessitant un traitement simultané de grandes quantités de données. CUDA (Compute Unified Device Architecture), développé par NVIDIA, permet de programmer efficacement ces unités en exploitant leur parallélisme intrinsèque.

Dans le cadre de ce travail, nous avons étudié la programmation parallèle avec CUDA à travers un problème réel : la génération de l'ensemble de Mandelbrot. Cet ensemble, célèbre pour sa complexité fractale, est un défi computationnel en raison de la quantité d'opérations répétitives nécessaires pour déterminer l'appartenance de chaque point à l'ensemble.

Le rapport est structuré comme suit : d'abord, nous présentons un rappel des concepts mathématiques liés à l'ensemble de Mandelbrot. Ensuite, nous détaillons l'algorithme séquentiel utilisé. Puis, nous analysons les potentiels de la parallélisation. Après cela, nous expliquons notre stratégie de parallélisation et décrivons la solution parallèle mise en œuvre. Enfin, nous comparons les performances entre la solution proposée, le code séquentiel et celui réalisé avec la programmation parallèle utilisant Pthreads, afin de souligner les gains significatifs obtenus grâce à la programmation sur GPU.

## 2. Objectifs

Les objectifs de ce travail pratique sont :

1. Identifier les éléments de parallélisme dans un programme séquentiel donné, en vue de son exécution sur une architecture GPU.
2. Utiliser effectivement CUDA pour paralléliser un programme séquentiel donné, en vue d'une exécution sur une architecture de type GPU de NVIDIA.
3. Comparer le temps d'exécution parallèle d'un programme donné, exécuté sur une architecture multicoeur et sur une architecture GPU de NVIDIA.

### 3. Rappel du problème traité

L'ensemble de Mandelbrot, découvert dans les années 1980 par le mathématicien Benoît Mandelbrot, est une fractale célèbre en mathématiques. Il est défini comme l'ensemble des points  $c \in \mathbb{C}$  pour lesquels la suite récurrente suivante reste bornée :

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

Il est prouvé mathématiquement que si la valeur du module  $|z_n| \geq 2$ , la suite va diverger vers l'infini, et le point  $c$  sera alors considéré comme étant en dehors de l'ensemble de Mandelbrot.

Pour représenter cet ensemble visuellement, on parcourt une région du plan complexe, chaque point de cette zone représentant un complexe  $c$ . On applique l'équation de récurrence définie précédemment pour chaque point, jusqu'à ce que la suite diverge ( $|z_n| \geq 2$ ) ou jusqu'à un nombre maximal d'itérations. Si ce nombre maximal est atteint sans que  $|z_n|$  dépasse 2, on conclut que la suite est bornée et donc le point  $c$  appartient à l'ensemble. Chaque pixel de l'image représente un point  $c$  et est coloré en fonction du comportement de la suite récurrente :

- Les points qui appartiennent à l'ensemble de Mandelbrot sont colorés en noir.
- Les points en dehors de l'ensemble sont colorés en fonction du nombre d'itérations nécessaires avant que la suite diverge.

Une illustration de cette représentation visuelle de l'ensemble de Mandelbrot est fournie dans la figure ci-dessous, mettant en évidence les points de divergence rapide en couleurs vives et les points appartenant à l'ensemble en noir.

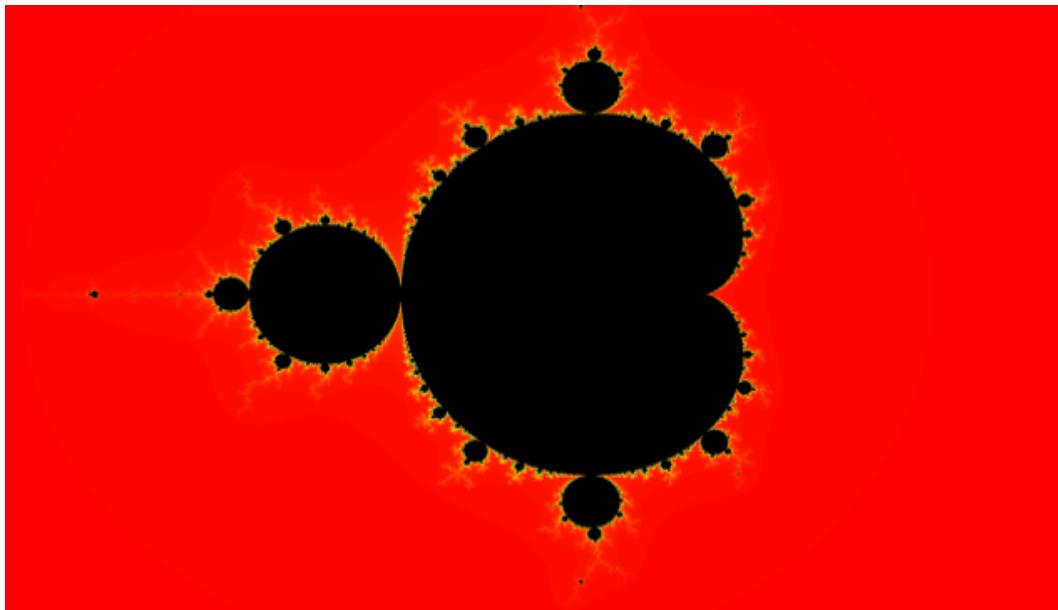
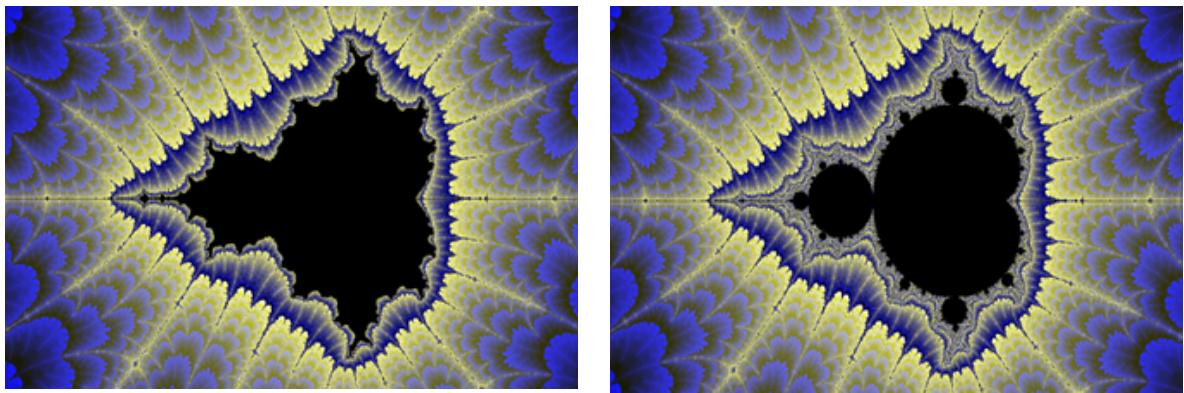


Figure 1: Représentation visuelle de l'ensemble de Mandelbrot.

L'une des propriétés fascinantes de l'ensemble de Mandelbrot réside dans le fait qu'il génère une fractale infiniment complexe : chaque fois qu'on zoome sur les bordures de sa représentation, on découvre de nouvelles structures fractales, de plus en plus petites, mais similaires dans leur apparence. Ce phénomène d'autosimilarité crée une beauté visuelle unique qui explique son attrait dans l'art génératif et la visualisation mathématique. Pour explorer davantage cette propriété fascinante de l'ensemble, on vous recommande de consulter le [lien suivant](#).

La précision de cette représentation dépend de deux facteurs principaux : la résolution de l'image (le nombre de pixels) et le nombre d'itérations utilisé pour chaque point. Plus la résolution est élevée, plus les détails sont fidèlement rendus. De même, augmenter le nombre d'itérations permet d'identifier les points qui appartiennent à l'ensemble avec plus de précision, notamment dans les zones complexes où la limite de divergence est difficile à déterminer.

La figure ci-dessous illustre l'effet du nombre d'itérations sur la précision de la représentation lors d'un zoom profond sur une bordure de l'ensemble de Mandelbrot.



Représentation avec 1000 itérations

Représentation avec 10000 itérations

Figure 2: Impact du nombre d'itérations sur la précision de la représentation lors d'un zoom profond sur une bordure de l'ensemble de Mandelbrot.

En pratique, la génération de l'ensemble de Mandelbrot est très gourmande en calculs. Une optimisation par parallélisation est donc nécessaire pour produire des images haute résolution, en particulier lors des zooms profonds où chaque pixel et chaque itération ajoutent des détails cruciaux à la structure.

Après avoir présenté les concepts fondamentaux de l'ensemble de Mandelbrot, nous allons désormais aborder l'algorithme permettant de générer une représentation visuelle de cet ensemble.

---

### Algorithme 1 : Représentation de l'ensemble Mandelbrot

---

**Entrées :** *maxIterations*. // nombre maximal d'itération.

*Largeur et hauteur*. // dimensions de l'image.

**Sortie :** Représentation visuelle de l'ensemble de Mandelbrot.

1. **Pour** chaque point de coordonnées (*x* ; *y*) du plan délimité par les dimensions de l'image
  2.  $c_r \leftarrow x$
  3.  $c_i \leftarrow y$
  4.  $z_r \leftarrow 0$
  5.  $z_i \leftarrow 0$
  6. *iterations*  $\leftarrow 0$
  7. **Tant que** ( $z_r^2 + z_i^2 \leq 4$  ( $|z| \leq 2 \Leftrightarrow |z|^2 \leq 4$ )) et (*iterations* < *maxIterations*)  
// Vérifier l'appartenance.
    8.  $temp \leftarrow z_r$
    9.  $z_r \leftarrow z_r * z_r - z_i * z_i + c_r$
    10.  $z_i \leftarrow 2 * z_r * z_i + c_i$
    11. *iterations*  $\leftarrow$  *iterations* + 1
  12. **Fin Tant que**
  13. **Si** *iterations* = *maxIterations* // Le point appartient à l'ensemble.
    14.     **Couleur du pixel**  $\leftarrow$  noir
  15. **Sinon** // Le point n'appartient pas à l'ensemble.
    16.     **Couleur du pixel**  $\leftarrow$  couleur en fonction du nombre d'itérations
  17. **Fin Si**
  18. **Fin Pour**
-

## 4. Intérêt de la parallélisation

Comme expliqué dans le TP précédent, la majeure partie du temps de calcul est consacrée à déterminer, pour chaque point, s'il appartient à l'ensemble de Mandelbrot ou non. Si l'on appelle **calculate\_pixel()** la fonction qui, pour un point donné, détermine son appartenance à l'ensemble, alors le "hotspot" du code correspond à l'exécution répétée de **calculate\_pixel()** sur chaque point de l'image.

Cette analyse montre que le calcul des valeurs de pixels pour l'ensemble de Mandelbrot est une opération très coûteuse, nécessitant de nombreuses itérations pour chaque point. Cela en fait un excellent candidat pour une optimisation par parallélisation. En effet, chaque point peut être traité de manière indépendante, ce qui permet de répartir les calculs sur plusieurs coeurs du GPU. Cette approche tire parti des ressources de calcul parallèle, réduisant ainsi considérablement le temps d'exécution total.

## 5. Caractéristiques du GPU

Avant de discuter de la stratégie de parallélisation sur GPU en utilisant CUDA, il est d'abord nécessaire d'étudier le matériel sur lequel la parallélisation sera effectuée.

Nom Complet du GPU	NVIDIA GeForce MX250
Architecture	PASCAL
Nombre total de coeurs CUDA	384
Nombre de SMs	3
Nombre de coeurs CUDA par SM	128
Taille du Warp	32
Nb max de thread/block	1024
Nb max de block/SM	106
Nb max de thread/SM	2048
Dimensions max du bloc [x, y, z]	[1024, 1024, 64]
Dimensions max de la grille [x, y, z]	[ $2^{31}-1$ , $2^{16}-1$ , $2^{16}-1$ ]

Tableau 1: Tableau résumant les caractéristiques de l'architecture du GPU dans laquelle nous avons effectué la parallélisation.

En analysant la table de spécifications du GPU (obtenue en exécutant un programme CUDA, partagé avec ce rapport), on peut déduire les points suivants :

- Le nombre maximum de tâches pouvant être exécutées parfaitement en parallèle est de: **Nombre\_SM \* warp\_size = 3 \* 32 = 96**.
- Pour exploiter plus d'un seul SM, le nombre total de blocs de threads doit dépasser **106**, ou bien le nombre total de threads doit dépasser **2048**.

## 6. Stratégie de la parallélisation

Comme expliqué dans la section : "Faisabilité de la parallélisation", le problème de Mandelbrot se réduit à une tâche atomique - que nous appellerons `calcul_pixel()` - qui s'exécute indépendamment pour chaque pixel de l'image.

Dans le contexte de la parallélisation sur GPU, cela nous donne une idée sur la fonction kernel qui sera exécutée par chaque thread, ainsi que sur la stratégie de parallélisation adoptée, en plus d'autres techniques et concepts propres à CUDA que nous mentionnerons ci-dessous.

- **Type de parallélisation :** Nous avons opté pour un **partitionnement des données**, c'est-à-dire un découpage de l'ensemble des pixels de l'image (éléments de la matrice) en sous-ensembles. Chacun de ces sous-ensembles est assigné à un thread distinct.
- **Nombre de threads:** Dans notre cas, chaque pixel sera assigné à un thread logique. Ainsi, pour une image de résolution (ResX, ResY), le nombre total de threads sera égal à  $\text{ResX} \times \text{ResY}$ .
- **Fonction kernel :** Chaque thread exécutera la fonction Kernel `calcul_pixel()`, qui consiste à calculer la valeur correspondant au pixel en question.
- **Organisation des threads :** Pour permettre à chaque thread d'identifier le pixel qu'il doit traiter, nous avons choisi une organisation à deux dimensions : Les threads sont organisés dans des blocs à deux dimensions, et les blocs eux-mêmes sont organisés en une grille à deux dimensions, dans ce cas, les coordonnées du pixel correspondant à un thread sont directement déduites à partir des identifiants (x, y) du thread au sein de son bloc, ainsi que des identifiants (x et y) du bloc dans la grille.
- **Variables GPU :** Une matrice de taille  $\text{ResX} \times \text{ResY}$  sera copiée vers la mémoire du GPU afin que chaque thread puisse y stocker la valeur calculée dans la case correspondant au pixel. À la fin de l'exécution du programme, cette matrice sera copiée vers la mémoire du CPU afin de générer l'image finale représentant l'ensemble de Mandelbrot.

- **Constantes GPU** : Certaines valeurs constantes (hyperparamètres fixés au début du programme) seront nécessaires à tous les threads pour effectuer les calculs. Une zone de mémoire constante sur le GPU a donc été allouée pour permettre un accès partagé par tous les threads. Un exemple de ces hyperparamètres inclut : ResX, ResY, LargeurRectangle, CentreRectangleX, CentreRectangleY, etc.

## 7. Solution parallèle

---

**Algorithme 2 :** Fonction main // Cette fonction sera exécutée par le CPU.

---

**Entrées :** ResX, ResY // Résolution de l'image

n // Nombre de threads par bloc, ce nombre doit être un carré parfait.

### 1. Début

2. Allouer une matrice "**resultat**" dans la mémoire du CPU de taille ResX × ResY.
  3. Allouer une matrice "**matriceGPU**" dans la mémoire du GPU de taille ResX × ResY.
  4. Allouer les valeurs constantes nécessaires sur le GPU.
  5. Déclarer une structure block de taille ( $\sqrt{n}$ ,  $\sqrt{n}$ ).
  6. Déclarer une structure grid de taille (ResY/ $\sqrt{n}$ , ResX/ $\sqrt{n}$ ).
  7. Activer le timer pour mesurer le temps d'exécution.
  8. Lancer le kernel : **calculate\_pixel<<< grid, block >>>(matriceGPU)**.
  9. Synchroniser les threads.
  10. Arrêter le timer.
  11. Copier le contenu de "**matriceGPU**" vers la matrice "**resultat**".
  12. Libérer la mémoire du GPU allouée pour "**matriceGPU**".
  13. Générer l'image finale à partir de la matrice "**resultat**".
  14. Fin.
-

---

**Algorithme 3 :** Fonction calcul\_pixel // Représente le Kernel, sera exécuté par le GPU.

---

**Entrées :** nbIterationsMax // Nombre maximal d'itérations.

ResX, ResY // Dimensions de l'image.

matriceGPU [ResX, ResY] // Matrice pour stocker les résultats.

Autres hyperparamètres nécessaires pour calculer x et y.

**1. Début Kernel**

```
2.   i ← blockIdx.x × blockDim.x + threadIdx.x
3.   j ← blockIdx.y × blockDim.y + threadIdx.y
4.   x ← coordonnée calculée sur l'axe des abscisses à partir de j
5.   y ← coordonnée calculée sur l'axe des ordonnées à partir de i
6.   cr ← x
7.   ci ← y
8.   zr ← 0
9.   zi ← 0
10.  iterations ← 0
11.  Tant que ( $z_r^2 + z_i^2 \leq 4$  ( $|z| \leq 2 \Leftrightarrow |z|^2 \leq 4$ ) et (iterations < nbIterationsMax))
12.    temp ← zr
13.    zr ← zr2 − zi2 + cr
14.    zi ← 2 × zr × zi + ci
15.    iterations ← iterations + 1
16.  Fin Tant que.
17.  Si iterations = maxIterations
18.    matriceGPU [ i , j ] ← 0 // Le point appartient à l'ensemble ; on lui attribue
       la couleur noire.
19.  Sinon
20.    matriceGPU [ i , j ] ← couleur en fonction du nombre d'itérations
21.  Fin Si.
22. Fin Kernel.
```

---

Les schémas ci-dessous présentent la répartition des données sur les threads, en mettant en évidence l'organisation bidimensionnelle.

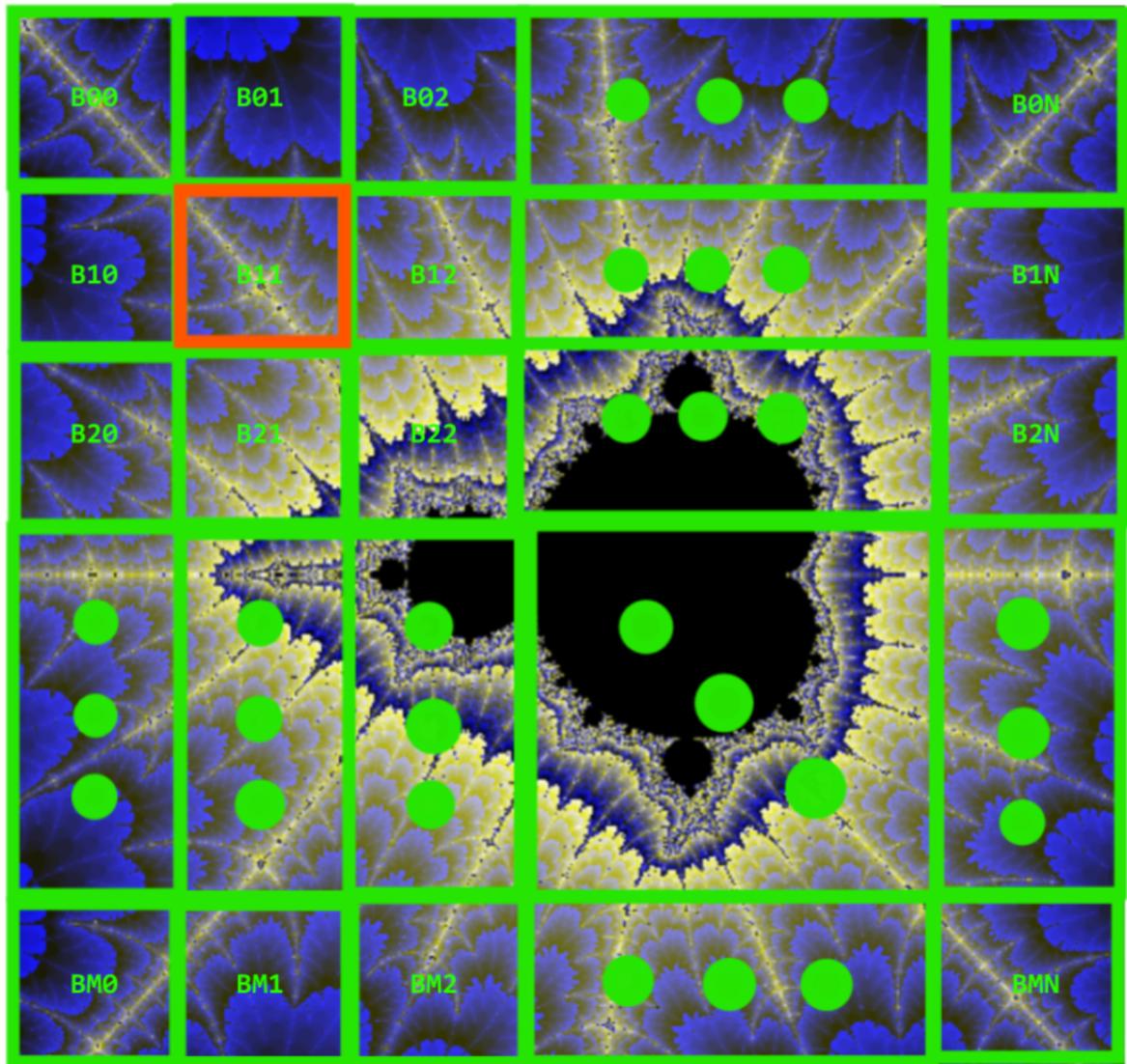


Figure 3: Organisation des blocs en une grille de taille  $M \times N$ .

En zoomant sur la partie entourée en orange, on peut observer l'organisation des threads dans le bloc B11. Chaque thread est assigné à un pixel, illustrant ainsi la répartition des tâches au sein du bloc.

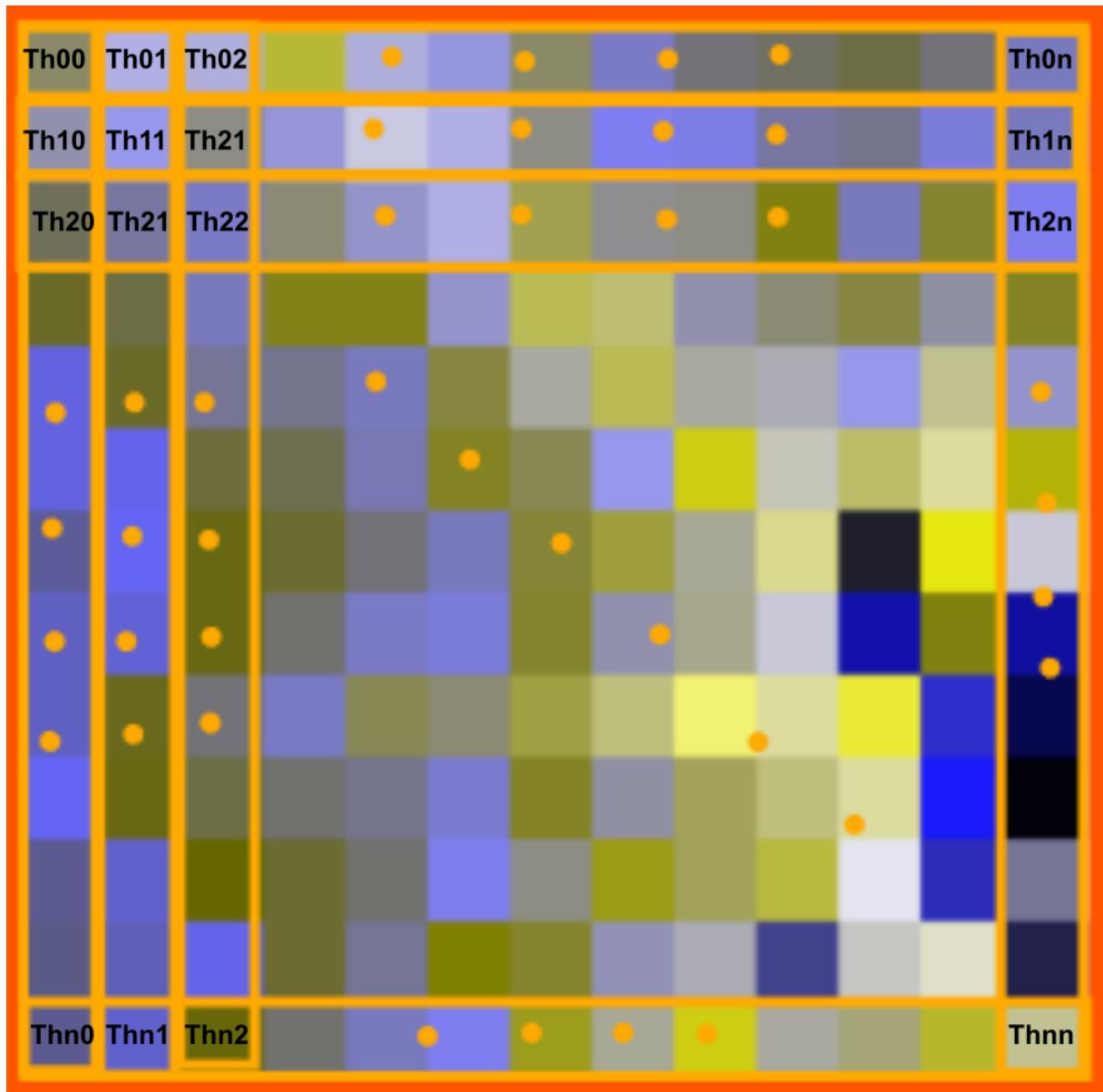


Figure 4: Organisation des threads dans un bloc  $n \times n$ .

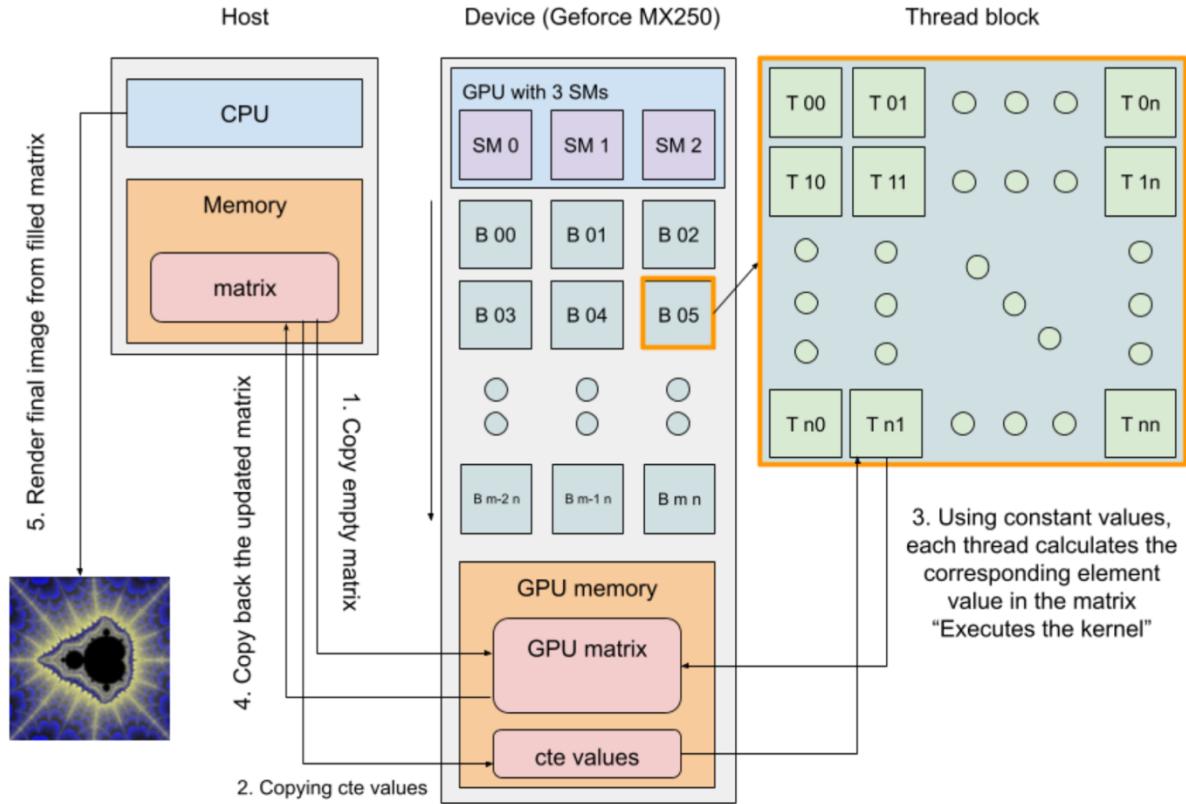


Figure 5: Schéma récapitulatif de la solution parallèle.

## 7. Résultats et interprétations

Cet exemple concerne la génération d'une image de résolution  $1024 \times 1024$  pour le cas de l'accélération GPU. En considérant notre solution proposée, le nombre de threads par bloc est de 1024, organisés en matrice  $32 \times 32$ , et le nombre de blocs est également de 1024, organisés en grille  $32 \times 32$ . Cela donne un total de  $1024 \times 1024 = 1\,048\,576$  threads créés. Avec un warp de 32 threads par SM et 3 SMs au total, on aura une parallélisation réelle de  $32 \times 3 = 96$  threads (ou 3 vagues de 32 threads chacune).

Méthode	Nombre de threads	Nombre maximal d'itérations	Temps d'exécution (s)	Accélération	Efficacité
Pthreads	<b>1</b>	1000	3.4	0,00	0.00
	<b>2</b>	1000	2.01	1.69	0.845
	<b>4</b>	1000	1.34	2.53	0.6325
	<b>8</b>	1000	1.10	3.09	0.38625
	<b>16</b>	1000	1.00	3.4	0.2125
	<b>32</b>	1000	1.18	2.88	0.09
CUDA	<b>1 048 576 en total - 96 en parallélisme réel -</b>	1000	0.225	15.11	0.15
Pthreads	<b>1</b>	10000	6.06	0,00	0.00
	<b>2</b>	10000	3.10	1.95	0.975
	<b>4</b>	10000	2.04	2.97	0.7425
	<b>8</b>	10000	1.33	4.55	0.56875
	<b>16</b>	10000	1.53	3.96	0.2475
	<b>32</b>	10000	1.48	4.09	0.127
CUDA	<b>1 048 576 en total - 96 en parallélisme réel -</b>	10000	0.668	9.07	0.1

Tableau 2: Tableau récapitulatif des mesures de performance entre la programmation parallèle Pthreads et CUDA.

La première remarque qu'on peut faire est que l'utilisation du GPU local donne un temps d'exécution beaucoup plus court par rapport à la meilleure performance obtenue avec pthreads, atteignant une accélération de 15 pour un nombre maximal d'itérations de 1000, et une accélération de 9 pour un nombre maximal d'itérations de 10 000. Cela s'explique par le fait qu'on change de matériel : avec pthreads, on utilise les 4 cœurs du CPU, tandis qu'avec CUDA, on utilise les 384 cœurs offerts par le GPU local. Cela montre que le GPU a un avantage en raison du nombre supérieur de cœurs ("threads") pouvant être exécutés en parallèle, soit  $32 \times 3 = 96$  threads, alors que le nombre maximal de threads pouvant être exécutés en parallèle sur un CPU est de 8.

En se basant sur l'efficacité, on peut en déduire que, de manière générale, un thread GPU est environ 4 fois moins efficace qu'un thread CPU (comparaison entre l'efficacité de CUDA et celle de pthreads avec 8 threads). Cela s'explique par le fait que les cœurs du GPU sont conçus pour des tâches simples et des calculs directs, qui sont moins complexes que ceux que les cœurs du CPU peuvent accomplir (comme les structures de contrôle telles que if/else, while, etc.). Si l'on analyse le code de la fonction kernel "calculate\_pixel()", on remarque qu'elle contient une boucle while avec un if/else, ce qui en fait une tâche plus complexe. Cependant, l'avantage du GPU réside dans le fait que le nombre de cœurs est bien plus élevé que celui du CPU et que le GPU est dédié au calcul, tandis que le CPU gère des tâches plus générales, comme l'exécution des différents programmes du système d'exploitation, des interruptions, etc.

## 8. Conclusion

Ce rapport a montré qu'il est possible de paralléliser efficacement un problème complexe, comme la génération de l'ensemble de Mandelbrot, en utilisant les GPU et la programmation parallèle avec CUDA. Grâce à leur capacité à gérer des milliers de threads simultanément, ces architectures offrent des accélérations significatives, rendant les calculs intensifs réalisables dans des délais réduits et ouvrant de nouvelles perspectives pour le traitement de problèmes computationnellement complexes.