ENSEIRB - MATMECA

PR310

Rapport de projet

BARACUDA : Affichage de fractales

Auteurs :
BODIN Antton
BRUNET Marion
CELEREAU Benjamin
THOMAS Sydney

Encadrants :
CRENNE Jérémie
LE GAL Bertrand

2020 - 2021

Sommaire

Première partie

Introduction

Ce projet de semestre a pour objectif une étude des algorithmes et méthodes numérique de calcul de fractales. Nous présenterons dans un premier temps l'objet d'études qu'est la fractale puis nous étudierons plusieurs techniques sur une fractale en particulier. Ceci nous amèneras enfin à étudier d'autres algorithmes et méthodes de coloration.

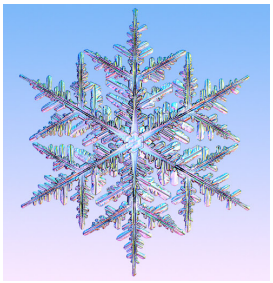
Les fractales sont des objets mathématiques découverts par Benoît Mandelbrot en 1975. Ces objets objets géométriques ont la particularité d'être auto-similaire, leur forme générale est constitué d'une version d'eux même à plus petite échelle. Ainsi lorsque l'on zoom on ne sait plus à quel échelle on se situe et on peut retrouver indéfiniment la même forme. C'est une version géométrique et visuelle de la notion de recursivité. un cas simple dans l'étude de la recursivité est le triangle de Sierpiński :

FIGURE 1 – triangle de Sierpiński

Comme on peut le voir ce dernier est constitué d'une forme triangulaire unitaire qui se répète à l'infini à l'image de poupées russes. Néanmoins les fractales ne sont pas seulement des objets abstraits ils existent naturellement.

(a) choux romanesco

(b) fractale de lichtenberg - courant surface bois



(c) cristal de glace

(d) fougère

FIGURE 2 – 4 exemples naturels de fractales

La raison de telles structures dans la nature peut s'expliquer par une nécessité d'optimisation. En effet les fractales optimise leur périmètre externe et donc la surface de contact tout en rationalisant le volume. Ainsi la fougère a accès à de l'air, de l'eau et de la lumière de façon optimale. Un autre cas explicite se situe simplement chez nous que ce soit notre système sanguin en ramification ou nos poumons. Ces derniers peuvent s'apparenter à des fractales et permettent d'optimiser le contact entre l'oxygène et le sang. Le concept mathématique bien que découvert récemment est simplement une modélisation d'un phénomène naturel du à une optimisation physique.

Si on s'intéresse ensuite aux applications de ces équations elles sont nombreuses et touchent à tous les domaines : physique, biologie, mécanique, géologie, médecine, astronomie mais aussi sociologie ou économie. Par exemple une des propriété remarquable de la fractale la plus connue, celle de Mandelbrot est son lien avec la suite logistique : $X_{n+1} = aX_n(1 - X_n)$. Cette suite non linéaire d'apparence simple permet notamment une modélisation de la taille d'une population en biologie.

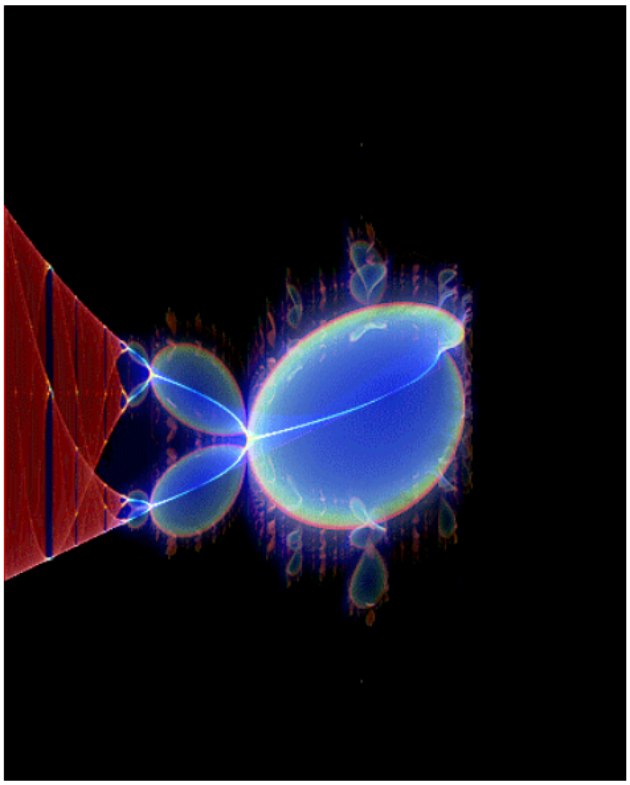


FIGURE 3 – Équation logistique superposée à la fractale de Mandelbrot dans un espace 3D

La fractale de Mandelbrot évoquée précédemment sera l'objet principal de notre étude. Cette fractale est la plus connue et est définie simplement. La fractale est définie par ce système :

$$\begin{cases} Z_0 &= 0 \\ Z_{n+1} &= Z_n^2 + C \quad \text{avec } C \in \mathbb{C} \end{cases} \tag{1}$$

Cela signifie que le premier terme de la suite est défini comme nul et chaque terme suivi est le précédent au carré plus une constante complexe. On effectue le calcul pour toute constante dans $\mathbb{C}$.

$$\begin{cases} \exists\, n \ |Z_n| & \geq & 2 & \text{alors } Z_n \text{ diverge} \\ \forall\, n \ |Z_n| & < & 2 & \text{alors } Z_n \text{ converge} \end{cases} \tag{2}$$

Ainsi on fait le calcul d'une convergence jusqu'à un N max. Si le module dépasse 2 pour un N on sait que le module au carré va augmenter exponentiellement donc on sait qu'il y aura divergence.

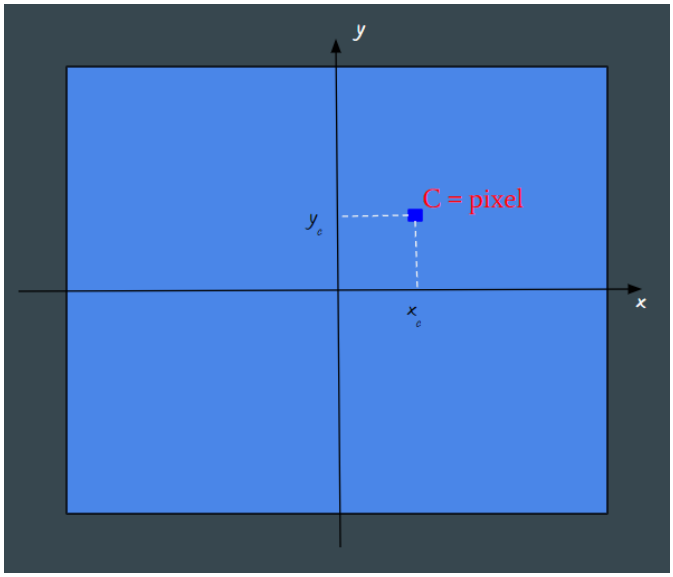


FIGURE 4 – e représentation d'un pixel sur le plan C

Afin de faire la représentation on va affecter la valeur d'un pixel à un complexe. La valeur variera seulement suivant le zoom. Cela nous permet de visualiser l'équation. Lorsqu'il y a divergence le pixel sera éteint et lorsqu'il y aura convergence le pixel sera coloré en fonction du n de la convergence.

Cet algorithme sera étudié principalement et comparé avec plusieurs technologies et méthodes de calcul. Nous nous concentrerons sur les performances pures mais nous ferons remarquer que cela aura un impact sur la qualité et/ou le coût du résultat.

Deuxième partie
CPU

La première solution envisagée, qui est généralement la plus courante pour développer un programme, est le **Central Processing Unit (CPU)**. Plusieurs raisons sont à l'origine d'un tel choix :

— ce composant est forcément disponible sur un ordinateur puisqu'il est l'un des **composants essentiels du PC**. Sans lui, pas d'ordinateur
— le coût d'un CPU est variable : de très faible à très élevé. Quoi qu'il en soit, même **la solution la moins chère permet d'exécuter un programme**
— c'est la solution la plus simple à prendre en main puisqu'il n'est pas particulièrement nécessaire d'anticiper la gestion des ressources, c'est pourquoi c'est ce support qui est **généralement utilisé pour apprendre**. Cependant ce n'est pas la plus efficace car non-dédiée

FIGURE 5 – CPU Intel Core i7

Avec cette solution que nous considérerons à bas coût, l'objectif sera, à partir d'une solution de base, d'*améliorer les performances* au travers de la vitesse de rafraîchissement de l'affichage tout d'abord *au dépend du coût*, puis *au dépend de la qualité*.

Pour être plus précis, par coût nous évoquons le *coût en surface silicium* impliqué dans l'utilisation de plusieurs ressources en parallèle dans le **multithreading** et le **SIMD** (termes explicités plus tard) ainsi qu'*énergétique* associé à leur alimentation. Par qualité, nous évoquons cette fois-ci la *précision des calculs* avec différents types de codage des données.

I Situation initiale

Une solution fonctionnelle initiale nous a été fourni par l'encadrant :

— le programme ouvre une interface graphique via l'API **Simple and Fast Multimedia Library (SFML)**
— l'affichage de la fractale de Mandelbrot est intégrée
— il est possible d'interagir avec l'interface en utilisant un clavier et/ou une souris pour se déplacer, (dé)zoomer ou changer certains paramètres du calcul tels que le nombre d'itérations maximum
— si une interaction a lieu avec l'un des périphériques cités, l'image est recalculée pour les nouveaux paramètres et/ou la nouvelle zone ciblée. Le temps nécessaire au calcul de la nouvelle image est indiqué en *millisecondes* dans le terminal associé

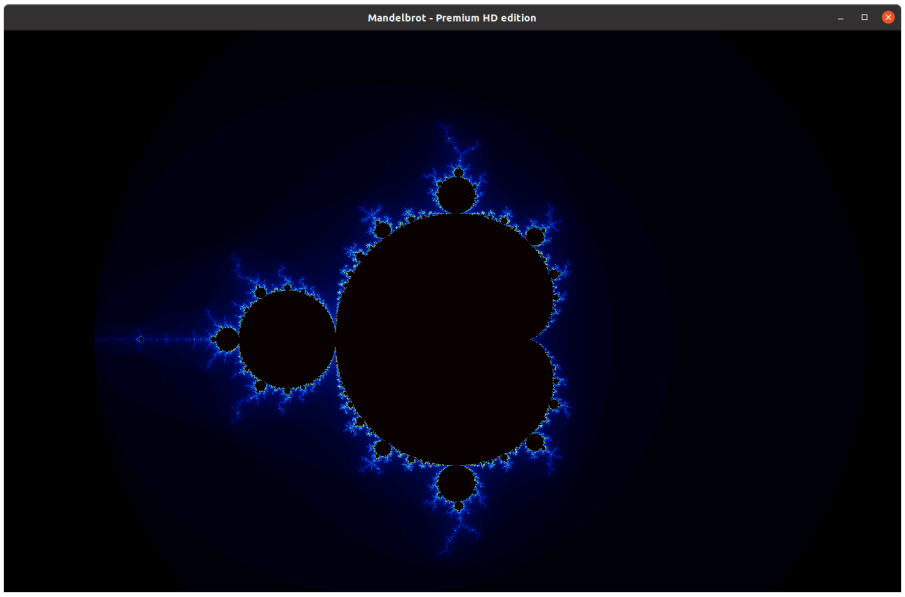


FIGURE 6 – Affichage initial de l'application

Le projet fonctionne correctement et nous pouvons déjà nous faire une idée des performances à partir des données affichées dans le terminal. Ci-dessous un graphique représentant l'évolution du temps nécessaire pour rafraîchir l'affichage selon le facteur de zoom ciblant une zone à fortes itérations.

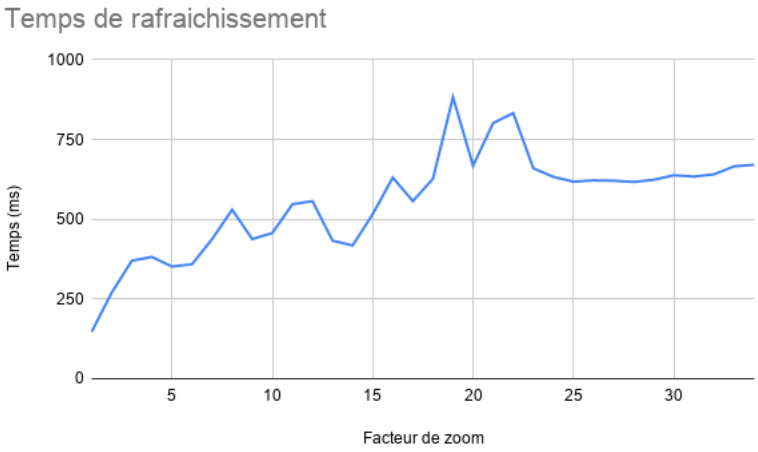


FIGURE 7 – Temps de rafraichissement de la solution initiale

En moyenne, **554 millisecondes** sont nécessaires pour rafraichir l'affichage variant entre **147** et **883 ms**, ce qui est relativement *lent*. Les parties suivantes illustrent les solutions mise en œuvre afin de diminuer le temps nécessaire au rendu.

II Optimisation avec OpenMP

La première optimisation mise en œuvre, parce qu'elle est très facile à utiliser, est **Open Multi-Processing (OpenMP)**.

Cette API a pour objectif de répartir les itérations de boucles équitablement entre les cœurs de calcul disponibles. Cette répartition est possible si et seulement si les plages de données sont indépendante. C'est-à-dire que les zones mémoires auxquelles on accède en écriture et/ou en lecture ne doivent par être les mêmes.

L'image suivante illustre la répartition des calculs dans le temps entre les différents cœurs. On observe *trois régions*, correspondant à 3 boucles consécutives, et dont le calcul est réparti sur *4 cœurs, puis 2 et 3*. Entre chaque région, on peut imaginer que d'autres opérations sont effectuées cette fois-ci sur un seul cœur.
Si OpenMP n'était pas utilisé dans ce modèle, le programme exécuterait les calculs de chaque thread de chaque région consécutivement sur un seul cœur, ce qui augmenterait sa charge et donc son temps d'exécution.

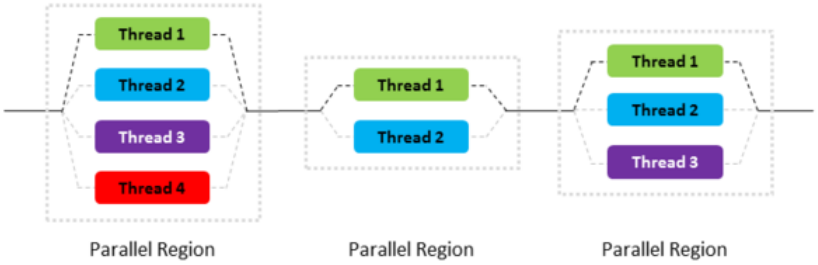


FIGURE 8 – Principe de fonctionnement de OpenMP

La parallélisation des calculs est particulièrement évidente avec le cas de notre problème. En effet, nous balayons l'entièreté de l'image d'abord par la hauteur, puis par la largeur. Le calcul de chaque pour chaque pixel se fait indépendamment de tous les autres, on peut donc utiliser OpenMP sur ces boucles.

```
for (int y = 0 ; y < IMAGE_HEIGHT ; y++) {
    for (int x = 0 ; x < IMAGE_WIDTH ; x++) {
        ...
    }
}
```

Par conséquent on peut par exemple répartir l'image, si on a *huit cœurs*, en *huit lignes horizontales équivalentes*.

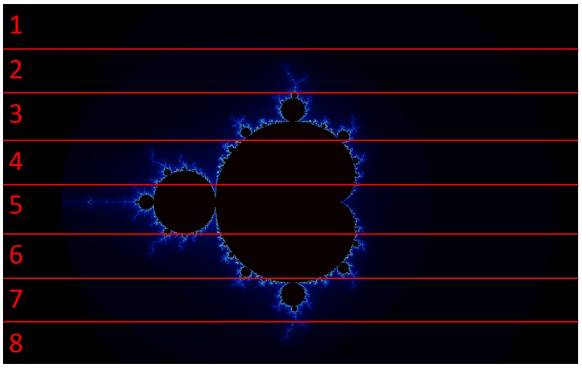


FIGURE 9 – Répartition de l'image avec OpenMP

Pour utiliser les instructions propres à OpenMP, on utilise des **pragmas**, des instructions propres au compilateur associé, afin de préciser que la boucle suivante devra être répartie entre les cœurs avec OpenMP :

```
#pragma omp parallel for num_threads(std::thread::hardware_concurrency()) schedule(dynamic)
for (int y = 0 ; y < IMAGE_HEIGHT ; y++) {
    for (int x = 0 ; x < IMAGE_WIDTH ; x++) {
        ...
    }
}
```

Dans ce pragma, on relève plusieurs mots-clés :

— `omp parallel for` : indique que la prochaine boucle "for" doit être divisée
— `num_threads()` : précise le nombre de threads à créer (idéalement le nombre de cœurs disponibles)
— `std::thread::hardware_concurrency()` : renvoie le nombre de cœurs disponibles sur le CPU
— `schedule(dynamic)` : répartition dynamique, permet de répartir les calculs dynamiquement pour qu'un cœur prenne en charge un thread dès qu'il est libre au lieu d'attendre inutilement que tous les threads aient traités tous leurs cas attribués

Idéalement, on souhaite diviser le temps total d'exécution par le nombre de cœurs disponibles. Ce n'est malheureusement pas possible parce que seuls les calculs sont répartis mais ils nécessitent une duplication de la mémoire partagée pour chaque cœur et l'affichage avec SFML n'est assuré que par un cœur.

Le graphique ci-dessous illustre les nouvelles performances (en rouge) comparées à celles obtenues sans notre outil (en bleu) :

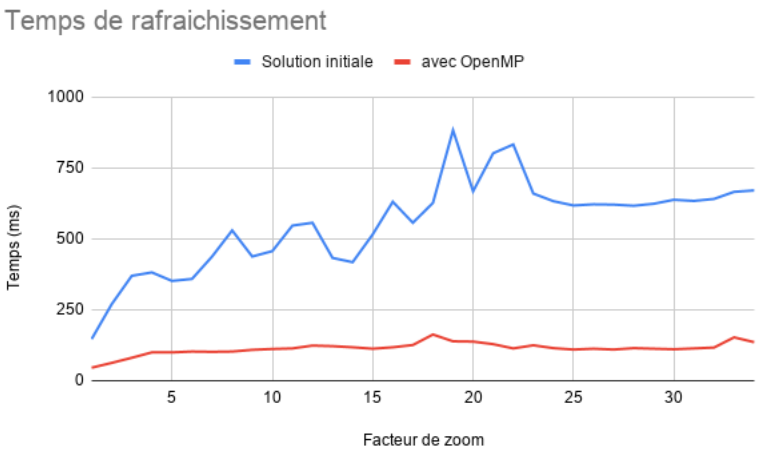


FIGURE 10 – Temps de rafraichissement avec OpenMP

Le temps moyen est maintenant de **113 millisecondes**, variant entre **46** et **163 ms**. Avec une machine à 8 cœurs, nous avons donc divisé notre temps par un **facteur 5** et équilibré le temps moyen, ce qui est considérable mais met également en évidence que le temps de copie de mémoire et d'affichage est considérable.

III Optimisation avec SIMD

Après avoir parallélisé nos calculs sur les différents cœurs et trouvé les directives permettant d'avoir la meilleure optimisation, il est possible d'aller plus loin. La parallélisation avec OpenMP se fait en fait sur un type d'architecture d'ordinateur qu'il est possible de changer. Il existe quatre catégories d'architecture d'après la taxonomie de Flynn proposée par Michael Flynn en 1966 qui sont classées selon le type d'organisation du flux de données et du flux d'instructions :

— SISD (unique flux d'instructions, unique flux de données)
— SIMD (unique flux d'instructions, multiples flux de données)
— MISD (multiples flux d'instructions, unique flux de données)
— MIMD (multiples flux d'instructions, multiples flux de données)

Pour expliquer chacune des architectures, les schémas suivants peuvent nous aider :

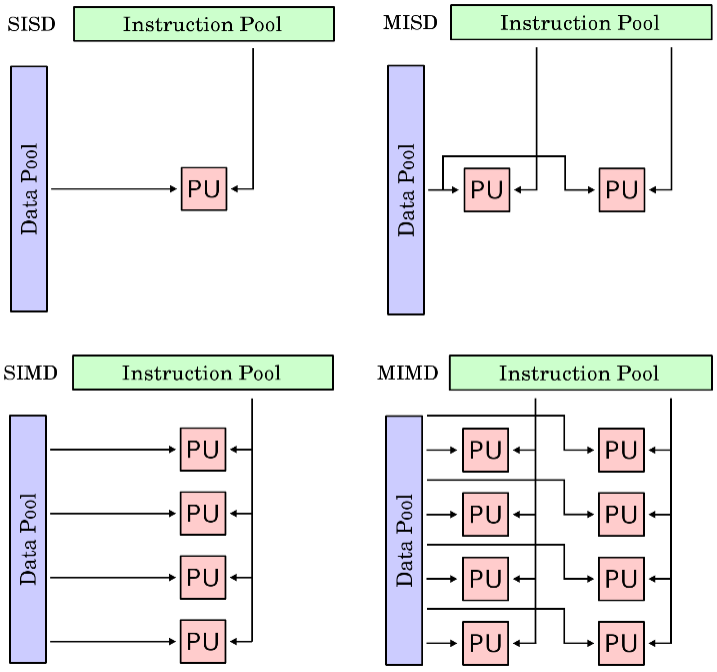


FIGURE 11 – Catégories d'architectures

avec ici :

— PU : unité de calcul (d'un processeur uni cœur ou multicœur)
— Instruction Pool : flux d'instructions
— Data Pool : flux de données

Ainsi, pour SISD, il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme, tant au niveau des instructions qu'au niveau de la mémoire. Cette catégorie correspond à l'architecture de von Neumann. Pour MISD, une même donnée est traitée par plusieurs unités de calcul en parallèle. Il existe peu d'implémentations en pratique. Pour SIMD, il s'agit d'un ordinateur qui utilise le parallélisme au niveau de la mémoire, par exemple le processeur vectoriel. Et enfin, pour MIMD, plusieurs unités de calcul traitent des données différentes, car chacune d'elles possède une mémoire distincte. Traditionnellement c'est le fonctionnement SISD qui est utilisé, nous voulons donc ici le changer pour passer en SIMD.

SIMD signifie donc Single Instruction Multiple Data. Dans ce mode, la même instruction est appliquée simultanément à plusieurs données pour produire plusieurs résultats comme on peut le voir sur la figure 12 :

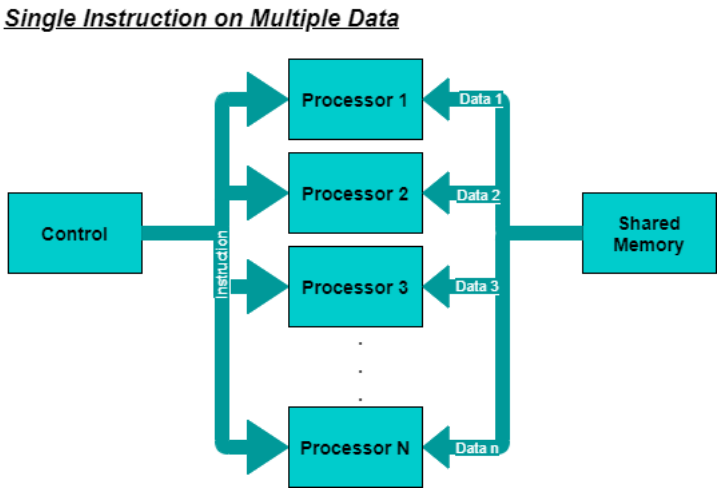


FIGURE 12 – Fonctionnement de SIMD

Comme expliquer plus haut, il est donc possible d'avoir différents résultats avec seulement une instruction, ainsi, les calculs peuvent se faire bien plus rapidement. En pratique, Le modèle SIMD est particulièrement utilisé pour les traitements dont la structure est très régulière comme c'est le cas pour le calcul matriciel. Les applications qui profitent généralement des architectures SIMD sont celles utilisant de nombreux tableaux, matrices ou structures de données du même genre. Dans notre cas, cela permet de parcourir l'image et faire les calculs plusieurs pixels par plusieurs pixels au lieu de faire chaque pixel un par un. Pour utiliser le modèle SIMD, il y a plusieurs possibilités :

— via l'utilisation d'instructions SIMD, généralement en microcode interprété sur du CISC ou câblé sur du RISC.
— par des processeurs vectoriels.
— par des processeurs de flux.
— via des systèmes comportant des processeurs multicœurs ou plusieurs processeurs.

Pour notre algorithme nous avons utilisé la première option c'est à dire l'utilisation d'instructions SIMD. Ces instructions nous permettent de faire plusieurs opérations de base en parallèle comme la soustraction, l'addition, la multiplication ou encore la comparaison qui nous seront très utile. Ces instructions SIMD peuvent se faire uniquement sur un ensemble de données de même taille et même type qui sont rassemblées dans des "blocs de données" d'une taille fixe nommé vecteur. Ces vecteurs contiennent plusieurs nombres entiers ou flottants placés les uns à côté des autres. Les instructions SIMD vont traiter chacune des données du vecteur indépendamment des autres et simultanément.

Les processeurs modernes contiennent des extensions à leur jeu d'instructions, comme le MMX et le SSE, etc. Ces extensions ont été ajoutées dans le but d'améliorer la vitesse de traitement sur les calculs. Les instructions SIMD sont ainsi composées de certains de ces jeux d'instructions. Pour notre projet, nous utilisons un processeur x86. Cette famille de processeur regroupe les microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086. Les jeux d'instructions possible pour SIMD sont donc ici MMX, 3DNow !, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2, AVX, AVX2 et AVX512. Les vecteurs traités par ces instructions sont placés dans des registres spécialisés dans le stockage de vecteurs. Nous avons utilisé, pour ce projet, des registres de 128 bits et 256 bits.

Pour transformer notre code avec ces instructions, il faut donc écrire des morceaux de code assembleur dans le programme pour profiter au maximum des optimisations permises par les instructions SIMD. Notre code de base utilisant des variables de type *double*, nous avons commencé par écrire un nouveau programme en utilisant un registre de 256 bits. Nous avons choisi celui-ci car il permet de stocker quatre doubles et donc de parcourir l'image assez rapidement :

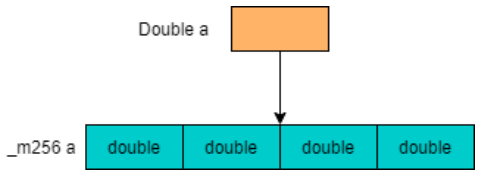


FIGURE 13 – Registre de 256 bits pour des doubles

Il faut donc modifier notre code en utilisant ce type de variables. La variable permettant de calculer les imaginaires devient donc un *_m256d* qui stock quatre fois la même valeur et la variable permettant de calculer les réels devient un *_m256d* qui stock quatre pixels différents. Cela permet de calculer quatre pixels par quatre pixels en fonction des x sur l'écran.

```
for (int y = 0; y < IMAGE_HEIGHT; y++) {

    double _startImag = offsetY - IMAGE_HEIGHT / 2.0 * zoom + (y * zoom);
    double _startReal = offsetX - IMAGE_WIDTH  / 2.0 * zoom;

    __m256d startImag = _mm256_setr_pd(_startImag, _startImag,
                                       _startImag, _startImag);
    __m256d startReal = _mm256_setr_pd(_startReal, _startReal + zoom,
                                       _startReal + 2.0 * zoom,
                                       _startReal + 3.0 * zoom);
```

FIGURE 14 – Programme avec des registres de 256 bits

La suite des calculs peut se faire grâce aux jeux d'instructions qui peuvent être trouver sur le site d'Intel (voir bibliographie). Une fois ce programme testé, nous pouvons le rajouter à la bibliothèque de convergence afin de pouvoir comparer les différents programmes facilement. Ensuite, nous avons décidé de changer le type des variables de base pour voir les différences de qualité et de rapidité. Nous avons donc changé les doubles par des flottants.

Pour rappel, voici comment sont codées les données décimales et comment sont attribués les différents bits pour ces deux formats :

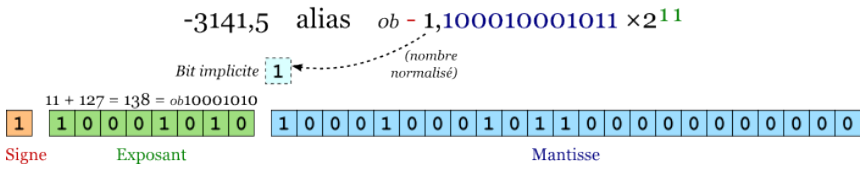


FIGURE 15 – Format de codage d'une donnée décimale

Type (taille)	Mantisse	Exposant	Signe
float (4 octets)	23 bits	8 bits	1 bit
double (8 octets)	52 bits	11 bits	1 bit

Les flottants étant plus petits, il est cette fois possible de stocker huit flottants dans le registre 256-bits ce qui permet de calculer huit pixels par huit pixels.

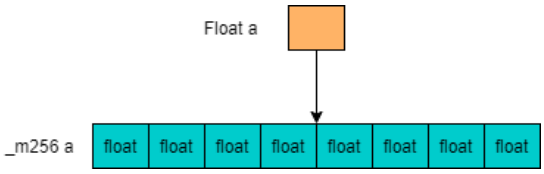


FIGURE 16 – Registre de 256 bits pour des flottants

Comme avec les doubles, il faut modifier le programme mais cette fois, les registres doivent stocker huit flottants comme on peut le voir sur la figure suivante :

```
for (int y = 0; y < IMAGE_HEIGHT; y++) {

    float _startImag = offsetY - IMAGE_HEIGHT / 2.0 * zoom + (y * zoom);
    float _startReal = offsetX - IMAGE_WIDTH  / 2.0 * zoom;

    __m256 startImag = _mm256_setr_ps(_startImag, _startImag, _startImag, _startImag,
                                      _startImag, _startImag, _startImag, _startImag);
    __m256 startReal = _mm256_setr_ps(_startReal, _startReal + zoom, _startReal + 2.0 * zoom,
                                      _startReal + 3.0 * zoom, _startReal + 4.0 * zoom,
                                      _startReal + 5.0 * zoom, _startReal + 6.0 * zoom,
                                      _startReal + 7.0 * zoom);
```

FIGURE 17 – Programme avec des registres de 256 bits

De même que pour le programme précédent, une fois testé nous l'avons ajouté dans la bibliothèque de convergences pour qu'il soit comparé aux autres. Pour finir sur la partie SIMD, nous avons décidé de tester un dernier programme SIMD en utilisant cette fois un registre 128-bits. Nous avons gardé des flottants car avec des doubles il n'est possible d'en stocker que deux ce qui ne suffit pas si l'on veut être plus rapide. Nous pouvons donc faire des calculs quatre pixels par quatre pixels.

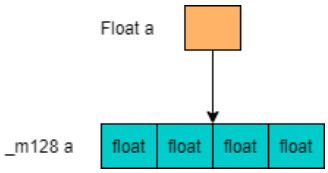
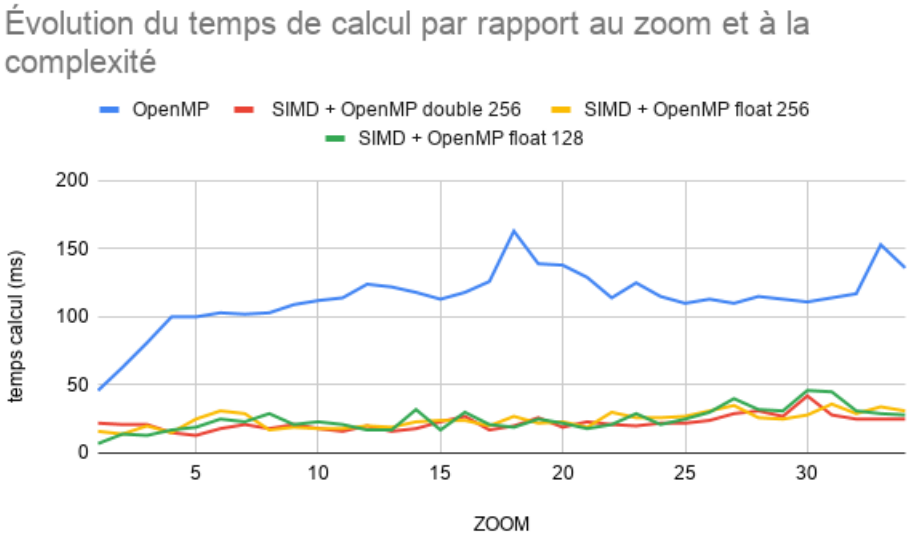


FIGURE 18 – Registre de 128 bits

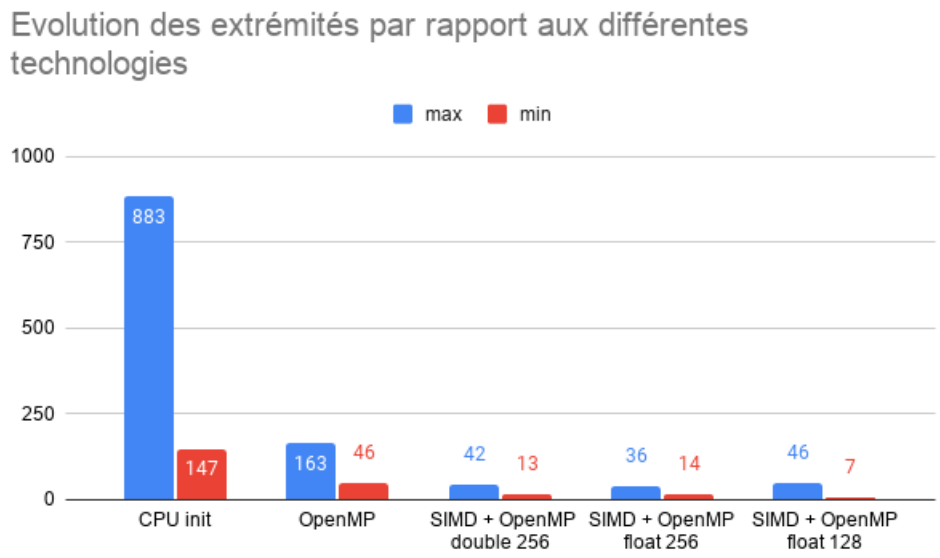
Le programme est alors modifié de la même façon que précédemment et nous pouvons le tester. Une fois testé, nous l'avons mis dans la bibliothèque de convergence. Nous pouvons donc maintenant comparer tous ces programmes pour voir lesquels sont les plus efficaces et voir si les changements ont réellement un impact par rapport au programme de base.

IV Bilan

Nous avons donc testé différentes optimisations en CPU et une bibliothèque permet de passer de l'une à l'autre facilement pour voir les avantages et inconvénients de chacune. Nous avons donc récupéré le temps de calcul en milliseconde de l'algorithme en zoommant un certains nombre de fois. Il est donc possible de voir l'évolution de chacune des convergences en fonction du zoom.



Comme le graphe le montre, l'évolution de tous les algorithmes est similaire, le temps de calcul est plus important lorsque le zoom est plus grand mais l'ordre de grandeur est toujours le même. Cependant, la courbe de l'algorithme avec uniquement OpenMP montre que le temps de calcul est bien plus important que pour les trois autres et que la différence entre sans zoom et avec un zoom important est beaucoup plus grande. Pour mieux pouvoir comparer entre chaque algorithme, le graphe suivant montre le temps maximum et minimal de calcul pour chaque algorithme :



Tout d'abord, le graphe montre bien que les optimisations sont vraiment efficaces, en effet, la différence du temps maximal et minimum entre l'algorithme initial et les autres est extrêmement importante. Ainsi, notre but d'optimiser l'algorithme est atteint. Maintenant, il faut trouver la meilleure optimisation possible. Les chiffres montrent, de plus, que l'utilisation de OpenMP uniquement n'est pas aussi efficace que l'utilisation de SIMD et OpenMP simultanément, ce qui est logique car SIMD permet de faire d'autant plus de calculs en parallèle et donc de diviser à nouveau le temps. Il est, par conséquent, plus intéressant d'utiliser OpenMP et SIMD ensemble bien que l'algorithme soit plus difficile à écrire car il faut écrire des morceaux d'assembleur comme nous l'avons vu plus tôt. Pour finir, il faut choisir entre les trois optimisations utilisant OpenMP et SIMD. Les chiffres étant très proches, il est bien de se concentrer à présent sur la qualité de l'image. En fait, même si le temps de calcul est un peu plus faible lorsque l'on utilise des flottants, l'image devient flou beaucoup plus rapidement et il n'est pas possible de zoomer autant qu'avec des doubles comme nous le verront plus tard. Ainsi, l'optimisation la plus intéressante, si nous voulons aller vite tout en ayant une image correct, est celle utilisant **SIMD et OpenMP avec un registre de 256-bits et des doubles**.

V Autres facteurs d'amélioration

Deux derniers facteurs peuvent être envisagés pour améliorer encore les performances. Nous n'avons pas souhaité les appliquer, chacun pour leurs raisons :

— Options de compilation :
Il est possible d'améliorer très facilement la vitesse de calcul des fonctions mathématiques avec les options `-O3`, `-Ofast` et `-ffast-math` du compilateur **GCC**. Cependant, les fonctions mathématiques qui sont introduites sont dites agressives, c'est-à-dire qu'elles sont beaucoup plus performantes mais dégradent la précision des calculs. Dans notre cas le but est de conserver une bonne précision afin de pouvoir zoomer le plus profondément possible dans la fractale en conservant les formes les plus petites. Il n'était donc pas envisageable d'utiliser ces options de compilation.

— Optimisation des algorithmes :
Le souci avec ce facteur est qu'il faut étudier plus en profondeur le code afin de voir si certains calculs pourraient être évités. Par exemple les zones noires correspondent aux zones pour lesquelles les suites ne convergent pas, donc pour lesquelles on effectue beaucoup de calculs inutiles au vu du résultat obtenu. Cependant cette étape demande beaucoup de temps et nous avions d'autres objectifs d'ordre esthétiques (présentés en troisième partie).

Troisième partie
GPU

I Optimisation avec CUDA

Une autre solution pour optimiser nos calculs est d'utiliser l'API développée par NVIDIA : CUDA. Cette API permet de programmer des GPU en C/C++ pour leur faire exécuter des calculs en parallèle sur les différents cœurs. CUDA dispose de plusieurs avantages par rapport à OpenMP car les threads CUDA sont plus légers. De plus, les architectures GPU sont pensées pour traiter ce type de configuration (exécution de calculs sur plusieurs threads).

Notre code est alors divisé sur deux composants : un *host* et un *device*. L'*host* correspond au CPU et le *device* au GPU. Le GPU n'a donc pas accès à la mémoire du CPU et inversement. Les sections de codes parallèle sont exécutées par des **kernel**. Un kernel est une fonction qui est identifiée par l'utilisation des triples chevrons lors de son appel. Un certain nombre de threads s'exécutent dans le même kernel.

L'objectif de la parallélisation avec CUDA est de faire exécuter les calculs de convergence de chacun des pixels par le GPU. Nous devons donc allouer de l'espace mémoire dans le GPU afin d'y stocker les valeurs de convergence de chacun des points de l'image. Pour cela, on fait appel à la fonction `cudaMalloc()` qui prend en argument un pointeur vers l'espace mémoire et une taille d'allocation requise en octets. Lorsque chaque valeur de convergence a été calculée, il faut alors recopier le tableau de valeurs, qui est en mémoire du GPU, dans la mémoire du CPU pour pouvoir l'afficher à l'écran. Nous utilisons alors la fonction `cudaMemcpy()` qui prend en paramètre la destination de la copie, la source de la copie, la taille en octets de données à copier et le type de copie. Le type de copie peut être : de *host* vers *device*, de *device* vers *host*, de *host* vers *host* ou de *device* vers *device*. Ici, on souhaite copier des données du GPU vers le CPU, c'est donc une copie *device* vers *host*.
Les fonctions de l'API CUDA retournent un message de type `cudaError_t` attestant de la bonne exécution ou non de la fonction. Il est important de vérifier à chaque appel de fonction le message retourné qui doit être "*cudaSuccess*" si tout s'est bien passé. Pour nous faciliter la tâche, des fonctions `CUDA_MALLOC()` et `CUDA_MEMCPY()` ont été écrites et permettant d'appeler les fonctions et de vérifier le message d'erreur.

Nous avons donc créé une nouvelle classe C++ *Convergence_ GPU* qui reprend les mêmes attributs et méthodes que la classe *Convergence_ dp_ x86*. La méthode `updateImage()` permet, ici, de :
- définir l'architecture du kernel (nous expliquerons cette partie plus loin dans ce rapport) ;
- allouer la mémoire nécessaire dans le GPU ;
- appeler le kernel GPU qui exécute les calculs ;
- copier les données du GPU vers le CPU et afficher l'image à l'écran.

Afin d'expliquer comment nous avons développé notre fonction kernel, il faut d'abord noter que dans un GPU les threads appartiennent à des blocs qui appartiennent eux-mêmes à des grilles comme le montre la figure 19.

Pour notre projet, nous souhaitons avoir un thread pour chacun des pixels de l'image. Nous avons arbitrairement choisi de prendre 256 threads par blocs organisés en deux dimensions donc 16 lignes de 16 threads chacune. La résolution finale de l'image étant de 1280×800 (soit 1 024 000 pixels) il faut alors 4000 blocs de 256 threads que nous avons donc organisés dans un grille de 80×50 blocs. Le nombre de blocs et de threads est demandé entre les chevrons lors de l'appel du kernel. Nous avons alors défini une variable *grid* décrivant la géométrie de la grille et une variable *block* décrivant la géométrie d'un bloc afin de pouvoir faire l'appel suivant :

```
kernel_updateImage_GPU<<<grid, block>>>(zoom, offsetX, offsetY, IMAGE_WIDTH, IMAGE_HEIGHT, deviceTab, max_iters)
```

Dans le kernel, nous devons alors repérer chacun des threads par un identifiant unique. Pour cela, nous devons d'abord repérer le bloc dont l'identifiant est :

$$blockID = blockIdx.x + (blockIdx.y \times gridDim.x)$$

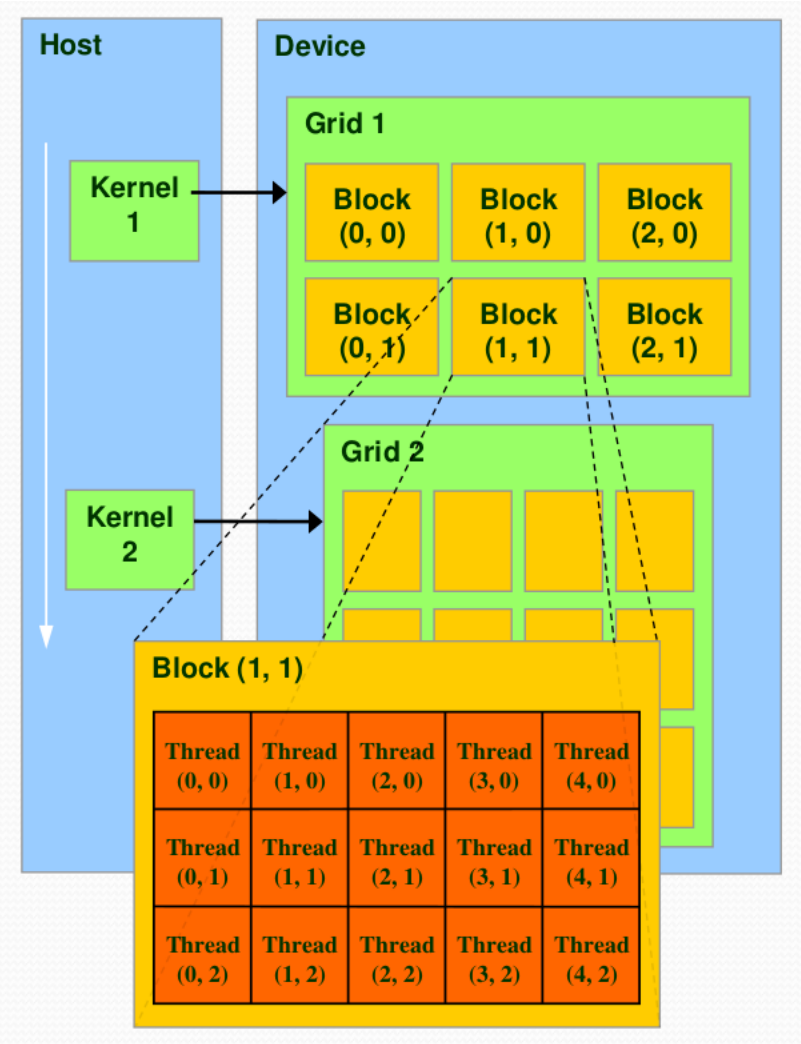


FIGURE 19 – Organisation des threads en blocs puis des blocs en grilles à l'intérieur d'un kernel

L'identifiant d'un thread dans le kernel est alors :

$$threadID = blockID \times (blockDim.x \times blockDim.y) + (threadIdx.y \times blockDim.x) + threadIdx.x$$

Ainsi, nous n'avons plus besoin d'utiliser des boucles `for()` car le kernel va être exécuté par le GPU pour chacun des 1 024 000 threads. Nous devons alors associer chacun des threads à une position en x et y dans le tableau de pixels. La position est calculée de la manière suivante :

$$x = threadID \ \% \ IMAGE\_WIDTH$$

$$y = threadID \ / \ IMAGE\_WIDTH$$

Le reste de l'algorithme est alors identique au calcul de la fractale de Mandelbrot sur CPU. Les valeurs de convergence obtenues sont alors stockée dans le tableau en mémoire du GPU à la position $y \times IMAGE\_WIDTH + x$.

II Bilan

La technologie par GPU a été testé avec un codage des données en floattant et en double. Comme lors des tests en CPU nous avons récupéré les métriques afin de comparer les résultats totaux. Tous les calculs ont été effectués sur la même machine afin de ne pas fausser les mesures.

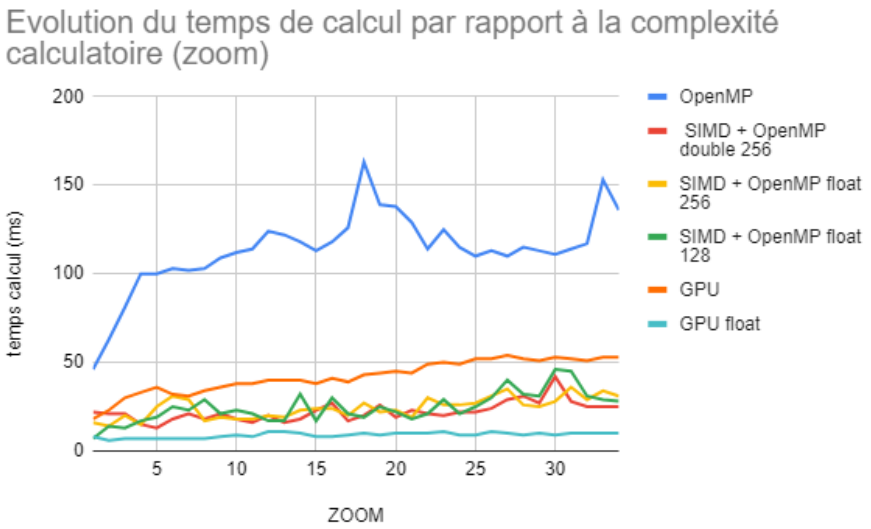


FIGURE 20 – Evolution des temps de calcul toutes technologies confondues

La figure 20 nous montre que le GPU peut concurencer le GPU mais ne se démarque pas beaucoup des améliorations déja apporté. De plus le GPU a une courbe d'évolution en augmentation nettement plus importante lorsque les données sont codées en double. Ceci signifie que pour des calculs complexes le GPU perd vraiment sont intérêt. On peut noter néanmoins que le GPU est très performant lorsque les données sont codées en flottant. En effet le GPU travaillant sur des flottant il est plus aisé pour lui de ne pas avoir de conversion à faire.

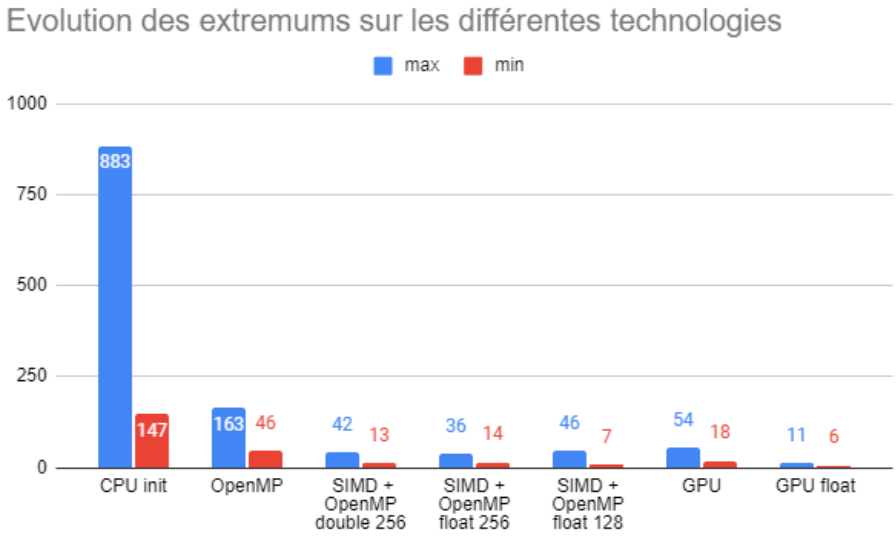


FIGURE 21 – Evolution des extremums toutes technologies confondues

Cette deuxième figure mesurant les extremums permet de voir que le GPU est vraiment au dessus de toutes les autres (hormis openmp) lorsque les données sont en double et en dessous de toutes les autres méthodes lorsque les données sont en float. Nous atteignons un maximum de performances avec le GPU dû aux échanges de données entre la carte graphique et le CPU. On peut envisager que les valeurs en float sont les performances maximales accessibles si on ne traite pas tout en GPU.

(a) mandelbrot codé double

(b) mandelbrot codé float

Ces dernières figures permettent de comparer le cas de données codées en double et en float. On voit une dégradation nette de la qualité en flottant. l'image est pixélisé rapidement mais permet des performances supérieures.

En conclusion le choix d'une méthodes est délicat car il correspond à un besoin. Que ce soit de performance de qualité ou de coût. Il convient de trouver un bon compromis. Dans notre cas seule la performance était prise en compte donc l'utilisation d'un GPU avec des flottants se révèle la meilleure solution. Mais au vu de la qualité fortement dégradé et du prix plus important face à un CPU seul, les autres solutions sont considérer. Enfin une solution sur FPGA pourrait être envisagé mais nous n'avons pas pus réaliser le comparatif par manque de temps.

Quatrième partie

Diversification

En dehors de l'optimisation des performances, comme avancé au cours de l'introduction, l'autre objectif majeur de ce projet était de diversifier le contenu. Nous allons expliciter dans cette partie les algorithmes de colorisation et les nouvelles fractales ajoutées.

I Colorisation

Pour la colorisation, nous avions un algorithme de base nous permettant de voir l'image montrée en introduction. Il est cependant possible de modifier l'algorithme pour obtenir d'autres couleurs ou d'autres affichages plus originaux. L'idée de colorisation d'une fractale est de colorer le pixel en fonction de l'itération à partir de laquelle la suite diverge. En fait, quand on sort de la boucle qui teste si la suite tend vers l'infini en module, on ne dessine rien, il faut donc dessiner le pixel avec une couleur en fonction du nombre d'itération nécessaires pour que la suite tende vers l'infini en module. Pour attribuer une couleur aux pixels, nous avons utilisé un code en RGB. Celui-ci est composé de trois nombres compris entre 0 et 255 qui représentent respectivement la quantité de rouge, de vert et de bleu. Il est donc possible de trouver différents algorithmes pour colorier différemment la fractale en fonction de ce qui a été expliqué juste avant. Ainsi, nous avons testé différentes colorisations et de même qu'avec les convergences, une bibliothèque existe et permet de passer d'une coloration à une autre facilement. Les figures suivantes montrent différentes possibilités de colorisation :

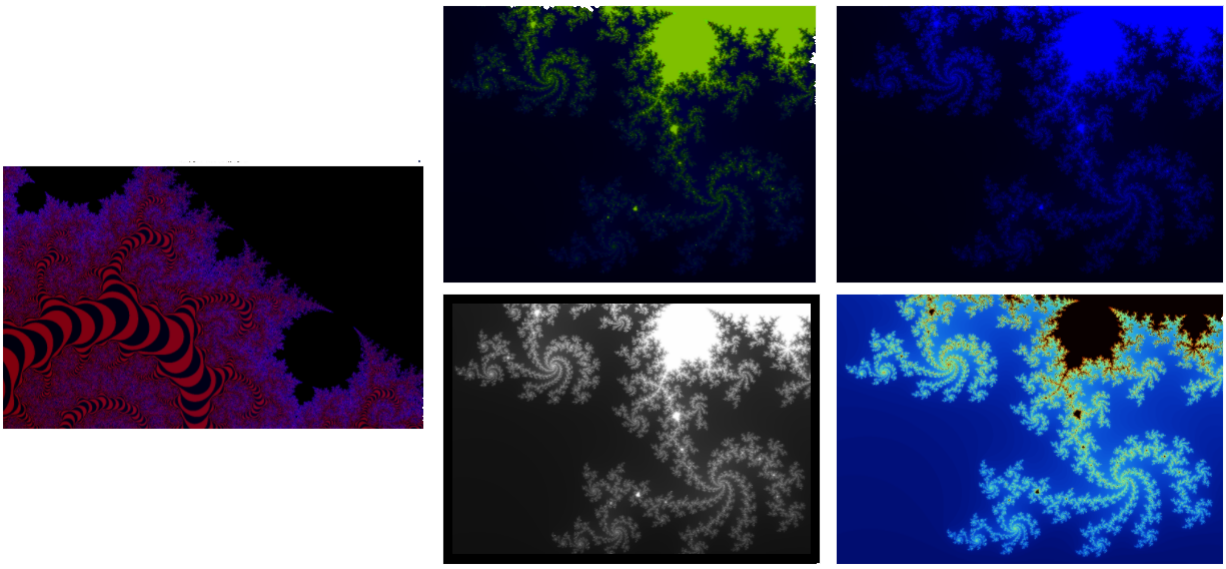


FIGURE 23 – Différentes map de couleurs

II Autres fractales

Nous avons pu intégrer plusieurs nouveaux algorithmes, toujours sur le même principe d'étude de la convergence des suites récursives de Mandelbrot. Nous allons illustrer ces différentes fractales.

1 ° Julia

La première fractale que nous avons ajouté est **Julia**, une déformation circulaire de Mandelbrot. À partir de ce dernier, on ajoute des facteurs a et b aux deux parties de la constante complexe c.

$$\left\{ \begin{array}{rcl} z_0 & = & 0 \\ z_{n+1} & = & z_n^2 + c \end{array} \right. \quad avec \quad c \quad = \quad a \times x_c + i \times b \times y_c \tag{3}$$

L'image suivante illustre le cas où $a = 0.285$ et $b = 0.01$:

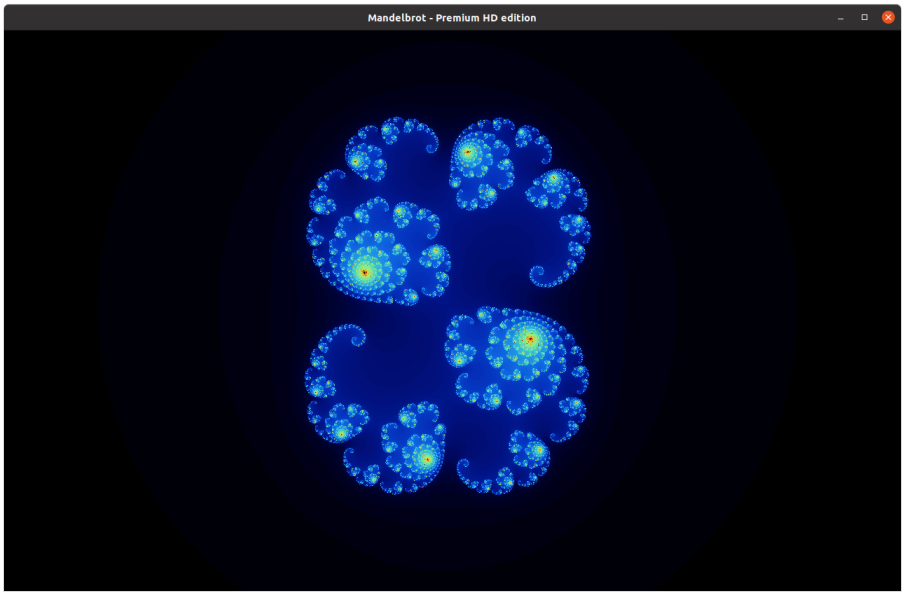


FIGURE 24 – Fractale Julia

Pour chaque fractale, nous déployons des solutions CPU double et M256D, et une solution GPU double.

2 ° Multibrot

Nous poursuivons avec Multibrot, qui multiplie les antennes de la fractale. Nous retrouverons donc plusieurs Mandelbrot répartis régulièrement autour d'un point central, tous dirigés vers l'extérieur.

La solution consiste simplement en la variation de la puissance de z_n dans la récurrence. Mandelbrot est le cas spécifique où $d = 2$.

$$\begin{cases} z_0 &= 0 \\ z_{n+1} &= z_n^d + c \end{cases} \quad avec \quad c = x_c + i \times y_c \tag{4}$$

Les deux illustrations suivantes sont celle où $d = 3$ et $d = -5$.

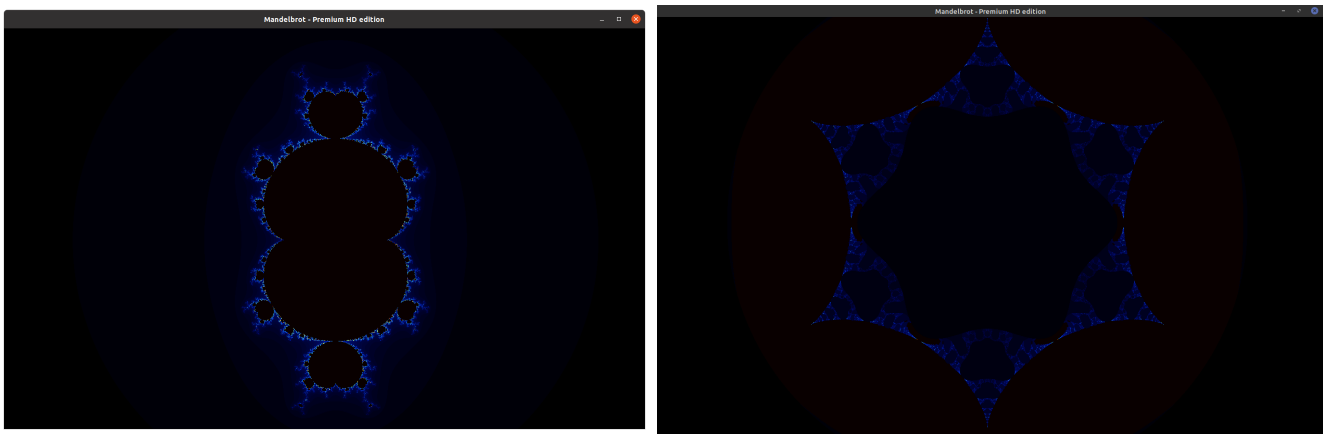


FIGURE 25 – Fractales Multibrot d=3 (gauche) et d=-5 (droite)

Le déploiement de cette fractale est plus compromettante pour le SIMD. En effet, pour une puissance définit, il suffit de développer l'équation de récurrence. Ci-dessous l'équation développée pour $d = 3$.

$$\begin{cases} x_{n+1} &= x_n^3 - 3 \times x_n \times y_n^2 + x_c \\ y_{n+1} &= 3 \times x_n^2 \times y_n - y_n^3 + y_c \end{cases} \tag{5}$$

En revanche, afin d'avoir un algorithme flexible en puissance, d'autres équations sont données :

$$\begin{cases} x_{n+1} &= pow((x_n^2 + y_n^2), \frac{d}{2}) \times cos(d \times atan2(y_n, x_n)) + x_c \\ y_{n+1} &= pow((x_n^2 + y_n^2), \frac{d}{2}) \times sin(d \times atan2(y_n, x_n)) + y_c \end{cases} \tag{6}$$

Un problème est induit dans ces deux équations par l'utilisation des fonctions `pow()`, `cos()`, `sin()` et `atan2()` qui ne sont pas disponibles en SIMD. Un autre problème créé par ces fonctions est leur faible rapidité, ce qui va à l'encontre de nos objectifs de performances. Ce point négatif peut toutefois être compensé par une approximation des fonctions avec des développements limités qui utilisent des fonctions usuelles.

3 ° Mandelbar

Cette nouvelle fractale diffère très peu de la précédente et présente les mêmes soucis de transcriptibilité et de rapidité que Multibrot. Ce qui diffère de cette dernière est l'utilisation de **l'inverse de** z_n montée à la puissance d.

$$\begin{cases} z_0 &= 0 \\ z_{n+1} &= \overline{z_n}^d + c \end{cases} \quad avec \quad c = x_c + i \times y_c \tag{7}$$

Ci-dessous la fractale pour $d = 3$:

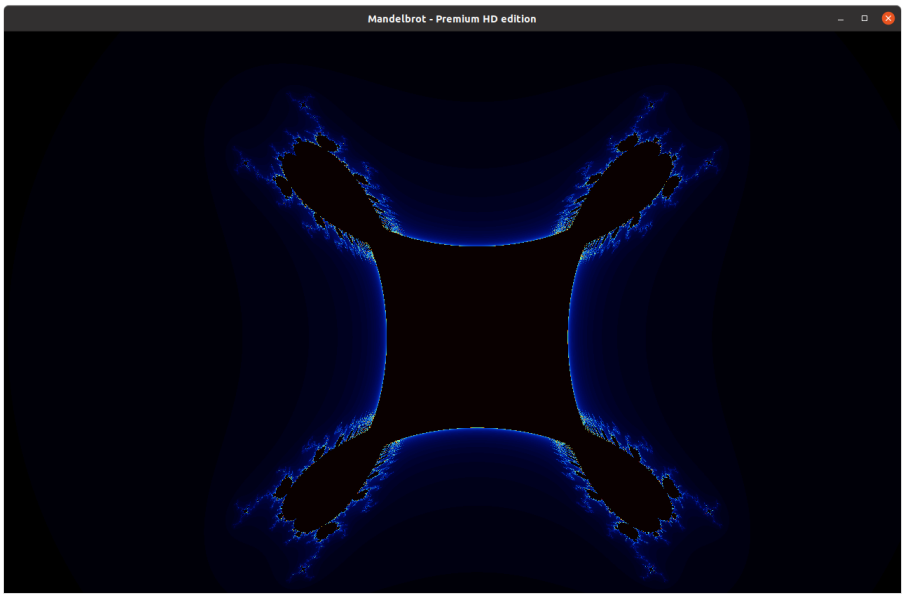


FIGURE 26 – Fractale Mandelbar

4 ° Monsieur et Madame

Deux fractales "remarquables" par leur ressemblance aux silhouettes d'un homme très musclé et d'une femme à forte poitrine sont *Monsieur* et *Madame*. Elles introduisent une nouvelle fonction lente et qui empêche la transcription en SIMD malgré le fait que l'on retire les fonctions précédentes : `abs()`.

$$\left\{ \begin{array}{rcl} z_0 & = & 0 \\ z_{n+1} & = & (|\Re(z_n)| - i \times \Im(z_n))^3 - i \times c \end{array} \right. \quad avec \quad c = x_c + i \times y_c \tag{8}$$

$$\left\{ \begin{array}{rcl} z_0 & = & 0 \\ z_{n+1} & = & (|\Re(z_n)| - i \times \Im(z_n))^5 - i \times c \end{array} \right. \quad avec \quad c = x_c + i \times y_c \tag{9}$$

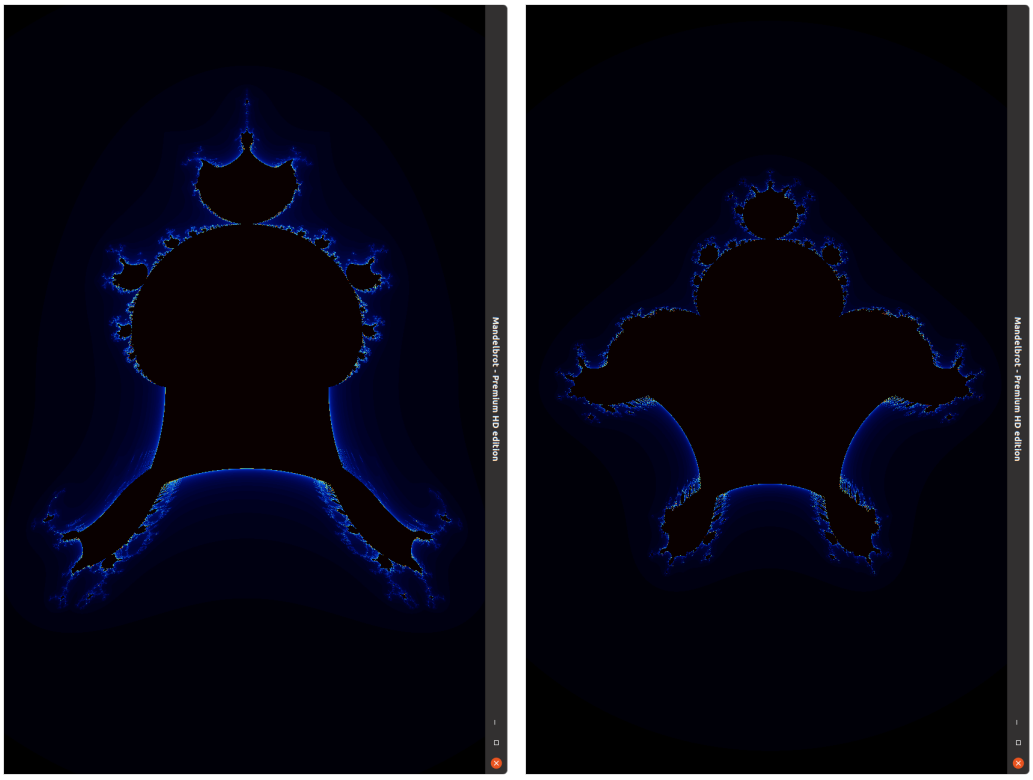


FIGURE 27 – Fractales Monsieur et Madame

5 ° Burning Ship

La dernière fractale mise en oeuvre, qui est la plus irrégulière mais sûrement la plus étonnante, est Burning Ship. Elle est très similaire dans l'équation à *Monsieur* et *Madame*.

$$\left\{ \begin{array}{rcl} z_0 & = & 0 \\ z_{n+1} & = & (|\Re(z_n)| + i \times |\Im(z_n)|)^2 + c \end{array} \right. \quad avec \quad c = x_c + i \times y_c \tag{10}$$

Elle produit alors la forme ci-dessous :

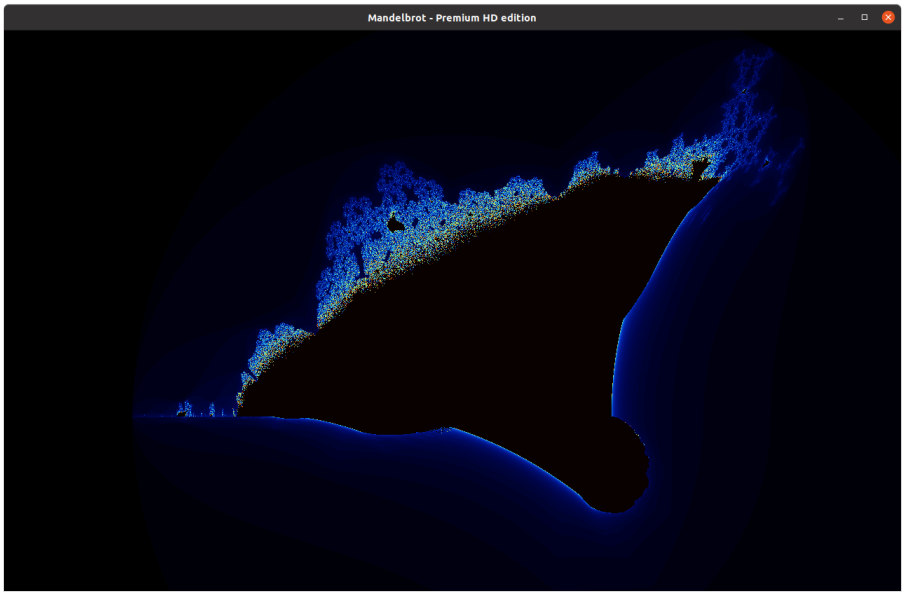


FIGURE 28 – Fractale Burning Ship

On peut identifier un paquebot enflammé en train de sombrer dans les eaux. En zoomant sur l'antenne inférieure gauche de l'image, on peut également obtenir la forme suivante, cette fois-ci assimilable à un galion aux voiles repliées.
Il est à noter que cette image n'est pas issu de notre programme parce que nous ne pouvions pas accéder à la zone où est située cette forme. En effet, nous pouvons nous déplacer et zoomer uniquement sur la diagonale haut-gauche à bas-droit. Nous avons essayé de solutionner le problème mais n'avons pas eu suffisament de temps pour cela.

FIGURE 29 – Fractale Burning Ship (zoom du l'antenne inférieure gauche)

Conclusion

Ce projet aura été l'occasion de comprendre les bases du calcul parallèle sur CPU et GPU. Pour cela nous avons utilisé différentes API et techniques telles que : OpenMP, SIMD ou CUDA. Le fait d'afficher une fractale à l'écran ainsi que le temps de calcul nécessaire pour chaque mise à jour de l'image était utile car il nous permettait de nous rendre compte en temps réel de l'impact de nos optimisations. Nous avons ainsi pu remarquer que même si le calcul était effectué plus rapidement, l'image était quant à elle dégradée. C'est le cas pour la calcul des convergences sur le GPU avec des données du type *float*.

Comme expliqué, nous avons du faire un compromis entre qualité et performance. L'idée étant d'optimiser le code au maximum tout en gardant l'image intacte et la possibilité de zoomer autant qu'on le souhaite, la solution qui a été retenue est celle faisant intervenir SIMD avec des registres de 256 bits sur des données codées en *double*.

Pour poursuivre ce projet, nous aurions aimé pouvoir ajouter d'autres types de fractales dans notre librairie. Ainsi, nous aurions aimé pouvoir tracer la fractale *Buddhabrot* qui reprend l'algorithme de Mandelbrot que nous avons présenté précédemment, tout en faisant intervenir des notions de probabilités. La fractale *Mandelbulb* fait quant à elle intervenir la 3D. Ces deux fractales auraient été intéressantes à étudier car il aurait été nécessaire de réfléchir à de nouvelles optimisations.

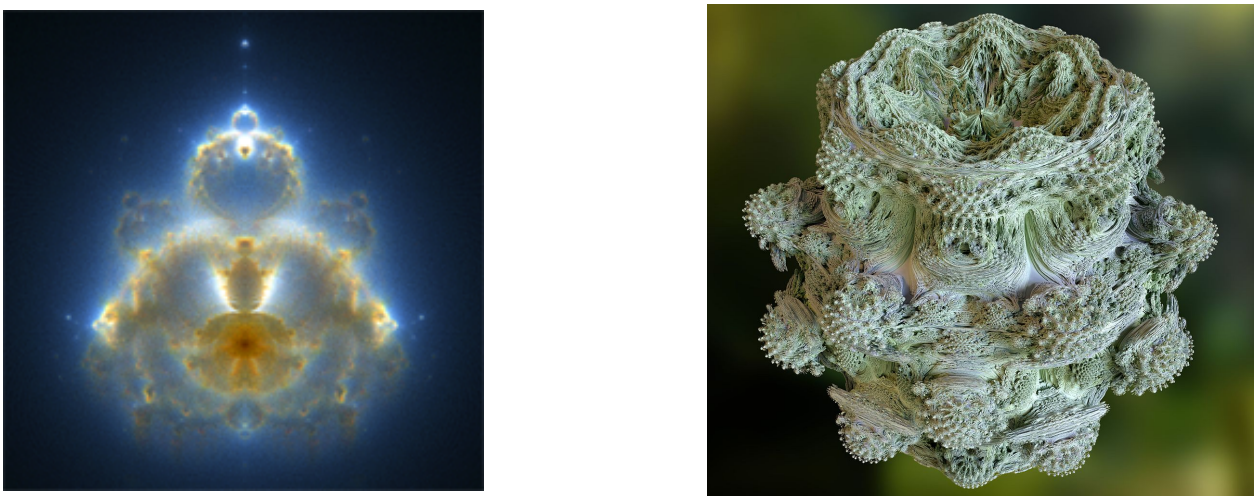


FIGURE 30 – Les fractales *Buddhabrot* (à gauche) et *Mandelbulb* (à droite)

Références

[Site internet] Wikipédia, Single instruction multiple data
https ://fr.wikipedia.org/wiki/Single_instruction_multiple_data

[Site internet] Intel Intrinsics Guide
https ://software.intel.com/sites/landingpage/IntrinsicsGuide/

[Site internet] Tutoriel : Dessiner la fractale de Mandelbrot
http ://sdz.tdct.org/sdz/dessiner-la-fractale-de-mandelbrot.html

[Site internet] Mathieu (2017), La fractale de Mandelbrot – Codage en Python
https ://mathete.net/la-fractale-de-mandelbrot/

[Site internet] Tutoriel OpenMP - Inria
http ://people.bordeaux.inria.fr/coulaud/Enseignement/PG305/pg305-OpenMP_2016.pdf

[Site internet] Introduction à OpenMP - Telecom SudParis
http ://www-inf.telecom-sudparis.eu/COURS/CSC5001/new_site/Supports/Cours/OpenMP/csc5001-openmp.pdf

[Site internet] Mandelbrot et ses potes
http ://eljjdx.canalblog.com/archives/2008/08/23/10295349.html

[Site internet] Intel IACA
https ://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html

[Site internet] Mandelbulb
http ://images.math.cnrs.fr/Mandelbulb.html