Rapport de projet

PR310

3e année électronique

Projet avancé de systèmes embarqués
Tracé de fractales

Auteurs :
Jonathan Saussereau
Denis Le Hégarat
Augustin Huet
Florian Loupias

Professeurs encadrants :
Bertrand Le Gal
Jérémie Crenne

Table des matières

1 Introduction **3**

2 Généralités **4**
2.1 Les fractales 4
2.2 L'ensemble de Mandelbrot 4
2.2.1 Algorithmie 5

3 Approche logicielle **6**
3.1 Programmation sur CPU uniquement 6
3.1.1 Implémentation en virgule flottante 6
3.1.1.1 Mise en œuvre simple de l'algorithme 6
3.1.1.2 Parallélisme avec OpenMP 6
3.1.1.3 Parallélisme SIMD et vectorisation avec AVX 7
3.1.1.4 Amélioration du parallélisme SIMD 10
3.1.2 Implémentation en virgule fixe 12
3.1.2.1 Généralités et mise en œuvre dans l'algorithme 12
3.1.2.2 Parallélisme avec OpenMP 12
3.1.2.3 Parallélisme et vectorisation avec SSE 13
3.2 Synthèse des résultats 14
3.3 Programmation sur GPU 15
3.3.1 La bibliothèque CUDA 15
3.3.2 La théorie sur le fonctionnement des GPU Nvidia 16
3.3.3 Le choix du type de variable utilisée 20
3.3.4 Optimisation 1 : calculs du meilleur nombre de threads par blocs . 21
3.3.5 Optimisation 2 : utilisation de double2 22
3.3.6 Optimisation 3 : fusion de fonctions 23
3.3.7 Optimisation 4 : utilisation de la Shared Memory 24
3.3.8 Optimisation 5 : utilisation de la Pinned Memory 24
3.3.9 Optimisation 6 : utilisation d'intrinsèques de calcul 25
3.3.10 Optimisation 7 : utilisation du type Float 25
3.3.11 Résultats et conclusion 27
3.4 Programme d'affichage 29
3.4.1 Affichage et entrées utilisateur 29
3.4.2 ColorMap 29
3.4.3 Fichiers de configuration 30
3.4.4 Enregistrement de mesures automatique 31
3.4.5 Paramètres d'exécution 32
3.4.5.1 Paramètres d'exécution généraux 32
3.4.5.2 Paramètres de convergence 33
3.4.5.3 Exemples de paramètres d'exécution 34

4 Approche matérielle **35**
4.1 Architecture 35
4.2 Gestion des entrées 36
4.2.1 Le zoom 36
4.2.2 Le mouvement 36
4.3 Calcul de la convergence 37

4.3.1 Synthèse de haut niveau HLS 37
4.3.2 Synthèse de bas niveau VHDL 37
4.4 Gestion de l'affichage 38
4.5 Optimisation - Passage à des variables sur 24 bits 38
4.6 Résultat 39

5 Conclusion **40**

A Annexes **41**
A.1 Approche logicielle 41
A.2 Approche matérielle 42

1 Introduction

Dans le cadre de la spécialité systèmes embarqués de troisième année de la filière électronique de l'ENSEIRB-MATMECA, nous avons été amenés à implémenter l'ensemble de Mandelbrot sur différentes technologies dans l'objectif de comparer celles-ci.

Ces implémentations ont été réalisées en sur processeur (CPU), carte graphique (GPU) et sur une architecture matérielle (FPGA). Le travail réalisé résume tout le processus d'implantation, d'optimisation et comparaison des performances. Embellir la fractale en y ajoutant de la profondeur et navigation dans celle-ci ont également été des critères importants permettant à ce projet d'approcher l'art fractal [1]

.

Supports de développement

Les différents supports sur lesquels ce projet a été réalisé sont les suivants :

— **pour l'approche logicielle sur CPU** : ordinateur muni d'un CPU Intel® Core™ i7-3610QM @ 2.30GHz x 8

— pour **l'approche logicielle sur GPU** : un ordinateur muni d'un CPU Intel® Core™ i5-7300HQ @ 2.50GHz x 4 et un GPU Nvidia GTX 1050 2GB

— **pour l'approche matérielle** : une carte composée d'un FPGA Xilinx Artix 7

1. Art fractal

2 Généralités

2.1 Les fractales

Les fractales sont des objets mathématiques, elles sont de multiples natures : courbes, surface ou même volume. Une fractale est définie par sa redondance et son invariance par changement d'échelle. Les fractales sont principalement des curiosités mathématique mais on retrouve des exemples de celle-ci dans le tracé des lignes de côtes ou l'aspect du chou romanesco.

FIGURE 2.1 – Un choux romanesco, souvent décrit comme un objet fractal naturel

2.2 L'ensemble de Mandelbrot

L'ensemble de Mandelbrot, du nom de Benoît Mandelbrot, est une fractale faisant partie de la famille des fractales à relation de récurrence. Cette fractale est le résultat graphique de la suite $z0 = 0$, $z_{n+1} = z_n^2 + c$ définie pour tout c dans $\mathbb{C}$. La suite converge pour tout c tel que $\exists\ n \in \mathbb{N} / |z_n| < 2$. On affiche alors en différentes couleurs le nombre d'itérations de la récurrence nécessaires pour avoir cette condition (ou, si le nombre maximal d'itération choisit est atteint, on affiche le point en noir).

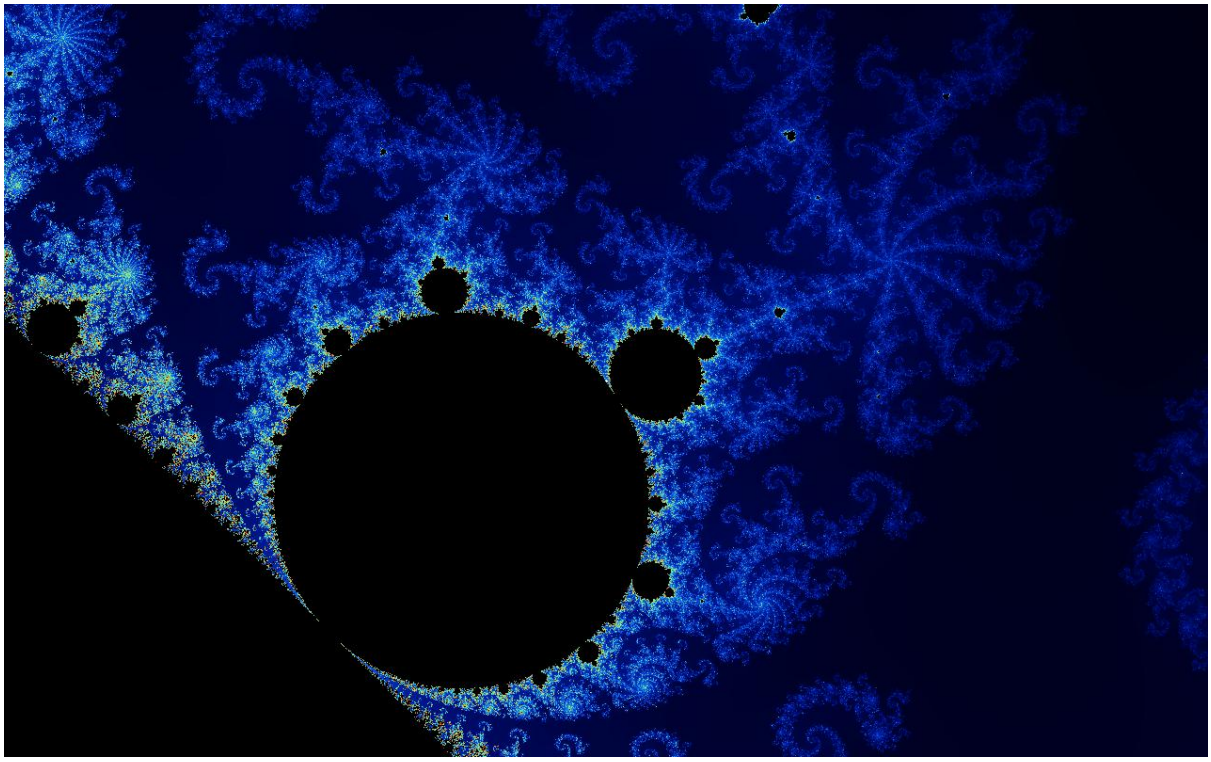


FIGURE 2.2 – Image d'une partie de l'ensemble de Mandelbort

2.2.1 Algorithmie

L' algorithme pour afficher l'ensemble de Mandelbrot est le suivant :

Pour chaque pixel P sur l'écran **faire** :
x, y coordonnées du pixel (à l'échelle)
r, i variables temporaires
Tant que $(r + i < 4)$ **faire** :
$rtemp = r - i + x$
$i = 2 \times r \times i + y$
$r = rtemp$
$itration = itration + 1$
Colorie P en fonction du nombre d'itération

3 Approche logicielle

3.1 Programmation sur CPU uniquement

3.1.1 Implémentation en virgule flottante

3.1.1.1 Mise en œuvre simple de l'algorithme

3.1.1.2 Parallélisme avec OpenMP

Pour réduire le temps de calcul, une première stratégie d'optimisation consiste à utiliser la totalité des cœurs de calculs du processeur plutôt qu'un seul. La bibliothèque OpenMP propose cette fonctionnalité.

En effet l'utilisation de plusieurs cœurs permet de paralléliser les calculs indépendants. Dans le cas du tracé de l'ensemble de Mandelbrot, chaque point (pixel) à tracer n'a aucune dépendance calculatoire avec les autres points. C'est donc un cas idéal pour le calcul parallèle. La directive `#pragma omp parallel for` permet d'indiquer au compilateur que la boucle `for` qui suit doit être parallélisée.

On peut cependant noter que l'utilisation de plusieurs coeurs de manière simultanée a des conséquences sur la fréquence d'horloge des cœurs. En effet dans certaines conditions, les processeurs récents peuvent augmenter leur fréquence de fonctionnement au delà de la valeur typique. C'est ce que l'on appelle le `Turbo Boost` chez Intel, activé en général que si un coeur uniquement est utilisé et que la température du CPU est suffisamment faible.

Sur lntel® Core™ i7-3610QM, les mesures effectuées ont montré une fréquence autour de 3.22 GHz en simple coeur mais une fréquence limitée à 3.092 GHz en multi-coeur. Cependant, malgré ce point, l'amélioration des performances reste significatives sur un applicatif fortement parrallélisable comme le tracé de l'ensemble de Mandelbrot, comme on peut le voir sur la figure ci-dessous.

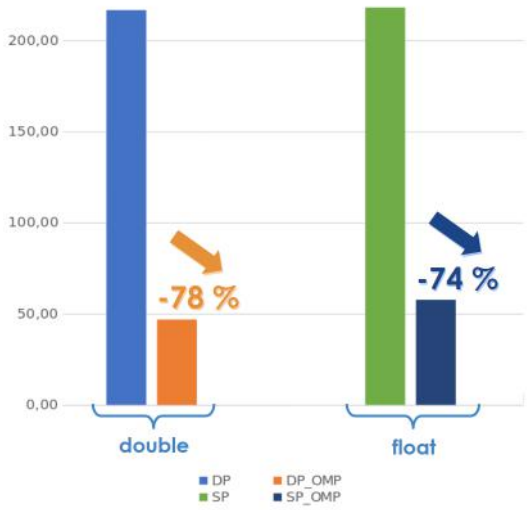


FIGURE 3.1 – Gain en temps d'exécution apporté par l'utilisation d'OpenMP [2]

2. Temps d'exécution médian obtenu pour 10 mesures successives sur le scénario "auto_zoom6", sur Ubuntu 18.04.3 LTS 64 bits avec un CPU Intel® Core™ i7-3610QM @ 2.30GHz x 8. Programme compilé avec clang 6.0.0-1ubuntu2 avec le flag -O3

On voit ici une amélioration par un facteur 4 du temps d'exécution. Il faut cependant noter la présence de 4 coeurs logiques en plus des 4 coeurs physiques sur le CPU de test, ce qui permet d'optimiser l'utilisation des ALU de chaque coeur. Une amélioration plus faible est à prévoir sur processeur à uniquement 4 coeurs physiques.

On peut noter sur la figure ci-dessus un résultat peu intuitif, la version à virgule flottante simple précision est plus lente que celle en double précision. En effet, comme on peut le voir en annexe A.1.1, pour l'architecture x86 Ivy Bridge, qui a été utilisée pour faire les tests, les multiplications sur les nombre flottants simple précision (32 bits) sont plus lentes que sur les nombres flottants double précision (64 bits). Sur d'autres architectures, comme ARM, ce n'est pas du tout le cas.

Il serait intéressant de rechercher qu'est ce qui architecturalement explique ces résultats.

3.1.1.3 Parallélisme SIMD et vectorisation avec AVX

La vectorisation consiste à empaqueter plusieurs données dans un seul registre de grande taille, comme on peut le voir dans la figure ci-dessous :

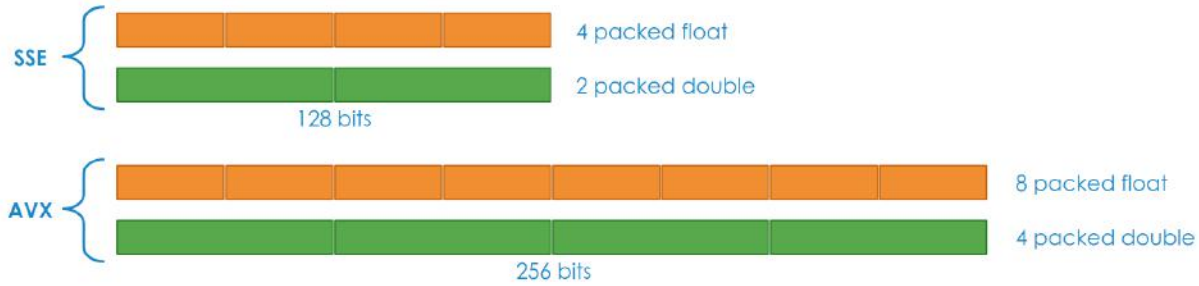


FIGURE 3.2 – Représentation des vecteurs SSE et AVX

L'intérêt d'utiliser des vecteurs de données plutôt que directement les données de manière scalaire est de réaliser une instruction sur chacune des données du vecteur en même temps. C'est ce que l'on appelle réaliser une instruction SIMD, SIMD voulant dire "Single Instruction on Multiple Data".

La vectorisation est réalisable sur les processeurs Intel/AMD x86 à l'aide des jeux d'instructions `SSE` et `AVX`. Sur processeur ARM, `ARM NEON` offre des fonctionnalités similaires.

La partie "implémentation en virgule flottante" se base sur l'utilisation d'AVX, qui propose des registres de 256 bits, permettant de stocker 4 doubles ou 8 floats.

Pour réaliser des opérations SIMD sur ces vecteurs, il faut utiliser les intrinsèques associées aux opérations désirées, disponibles **sur le site d'intel**.

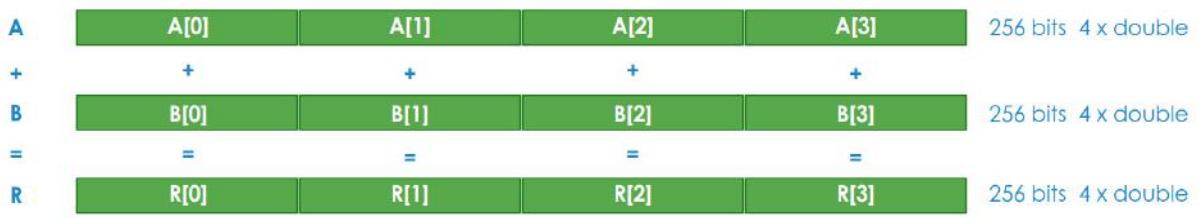


FIGURE 3.3 – Exemple d'une opération d'addition entre deux vecteurs

Dans le cas de la fractale de Mandelbrot, ce sont les mêmes calculs qui sont réalisés pour chaque pixel, mais avec des données d'entrées. Ainsi, les opérations SIMD peuvent être utilisées pour calculer la valeur de plusieurs pixels à fois.

Cependant, chaque pixel ne converge pas après le même nombre d'itération, il faut par conséquent attendre que le pixel à la convergence la plus lente termine. Traiter, dans le cas du calcul en double, 4 pixels à la fois ne permet donc pas d'aller 4 fois plus vite, mais permet de calculer 4 pixel dans le temps du pixel le plus long.

La principale difficulté apportée par le calcul SIMD dans le cas du tracé de l'ensemble de Mandelbrot est de vérifier que chaque pixel a finit de converger pour pouvoir passer aux pixels suivants.

On peut noter que lorsque l'on calcule la convergence d'un pixel, la valeur qui est comparée dans la condition de convergence (r^2+i^2) ne peut que décroître à chaque itérations. Ainsi, si l'expression est inférieure à une valeur à une itération, elle le sera aussi pour toutes les itérations suivantes. Cette propriété permet d'éviter d'introduire une notion de mémoire dans la vérification de convergence.

Il existe une intrinsèque de comparaison SIMD dans AVX nommée `_mm256_cmp_pd`, qui prend en paramètre le type de comparaison. La propriété précédente amène à opter pour une comparaison de type "inférieur ou égal". Cette intrinsèque de comparaison retourne un vecteur masque, dont les éléments dont à `0xFFFFFFFFFFFFFFFF` si la comparaison est vraie, ou à `0x0000000000000000` si la comparaison est fausse.

Une autre intrinsèque nommée `_mm256_movemask_pd`, permet de réduire ce masque à 4 bits contenus dans un entier, comme on peut le voir dans la figure ci-dessous.

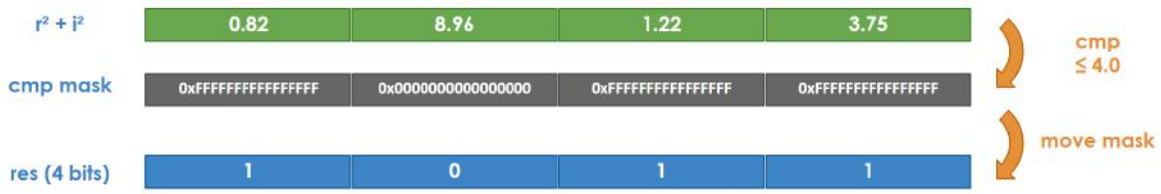


FIGURE 3.4 – Schématisation de la vérification de convergence pour tous les éléments

Ainsi, la convergence est atteinte pour tous les éléments si chaque bit de cet entier sont à 0. On a donc comme nouvelle condition de convergence la comparaison de cet entier à 0.

Il reste cependant un autre problème à résoudre. En effet, il faut stocker à quelle itération la convergence à été atteinte pour chaque élément. La solution retenue, consiste à incrémenter une variable jusqu'à ce que la convergence soit atteinte. Ce qui veut dire, compte tenu de la propriété précédemment énoncée, tant que $(r^2 + i^2)$ est supérieur à 4. Pour cela il est possible de réutiliser le masque obtenu pour la condition de convergence. Une autre intrinsèque nommée `_mm256_blendv_pd` permet de réaliser l'équivalent de l'opérateur ternaire du c `"condition ? valeur si vraie : valeur si fausse"`. Il prend en entrée deux vecteurs et un masque. Si la valeur du masque est `0x0000000000000000` pour un élément, alors la valeur de l'élément de même indice dans le vecteur de sortie sera la valeur de l'élément de même indice du premier vecteur et si la valeur est la valeur est `0xFFFFFFFFFFFFFFFF`, la valeur de l'élément de même indice dans le vecteur de sortie sera la valeur de l'élément de même indice du deuxième vecteur, comme on peut le voir

dans la figure ci-dessous.

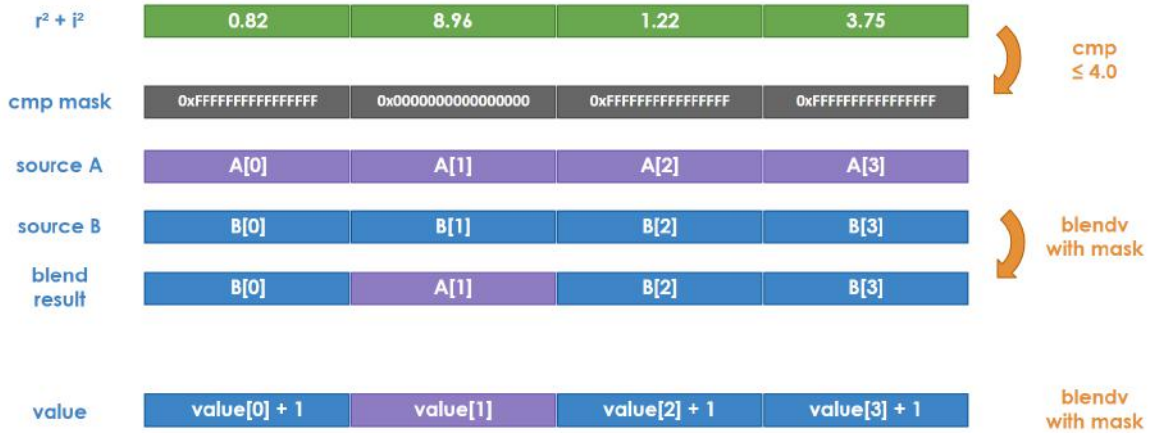


FIGURE 3.5 – Schématisation de l'incrémentation uniquement s'il n'y a pas eu convergence

On peut ainsi, utiliser comme deuxième vecteur le vecteur contenant le nombre d'itérations pour arriver à la convergence +1, et comme premier vecteur, ce même vecteur sans l'addition. Ainsi, le nombre d'itérations pour arriver à la convergence ne s'incrémentera que si la convergence n'a pas encore été atteinte.

L'utilisation des opérations SIMD a permit d'améliorer de manière significative les performances, comme on peut le voir dans la figure ci-dessous.

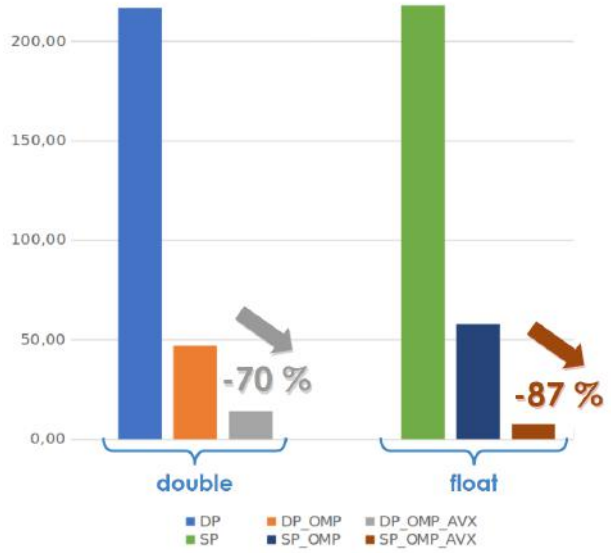


FIGURE 3.6 – Gain en temps d'exécution apporté par l'utilisation des opéra tions SIMD [3]

Plusieurs choses ressortent de ces résultats. Tout d'abord, il parait important de préciser que les résultats précédants ont été obtenus en utilisant le flag de compilation

3. Temps d'exécution médian obtenu pour 10 mesures successives sur le scénario "auto_zoom6", sur Ubuntu 18.04.3 LTS 64 bits avec un CPU Intel® Core™ i7-3610QM @ 2.30GHz x 8. Programme compilé avec clang 6.0.0-1ubuntu2 avec le flag -O3

`-O3`, qui permet au compilateur d'essayer d'utiliser comme il peut les unités de calcul SIMD. Le gain réél de l'utilisation des SIMD est donc potentiellement plus grand.

Ensuite, on remarque que le gain pour le calcul en float est plus important que pour la version double, et que celui en float est désormais plus rapide et même presque 2 fois plus rapide que celui en double.

Ces résultats sont parfaitement logiques et facilement interprétables. En effet, les deux implémentations utilisent exactement les mêmes opérateurs AVX, mais pour un format de donnée différent. Comme on peut le voir **sur le site d'intel**, l'ensemble des opérateurs utilisés ont les mêmes performances en simple et en double précision.

Cependant, en simple précision, 2 fois plus de données peuvent être traitées en même temps. Il faut bien sûr prendre en compte le fait que le temps d'exécution pour un vecteur n'est pas le temps moyen des éléments mais le temps maximal des éléments. Cependant, on imagine bien un facteur proche de 2 en temps d'exécution entre double et simple dans ces conditions, ce qui est bien ce que l'on retrouve.

3.1.1.4 Amélioration du parallélisme SIMD

La version SIMD précédemment présentée peut encore être améliorée. En effet, les opérateurs utilisés ont tous un bon débit, généralement autour d'un cycle par instruction mais on une latence qui peut atteindre 3 ou 4 cycles. Cela signifie que si l'on effectue une opération, même le cycle suivant l'unité de calcul est prête pour en faire un autre calcul, le résultat de la première ne sera disponible qu'après quelques cycles. Il est donc important de surveiller la dépendance des données, pour éviter d'avoir des cycles sans calcul.

Pour utiliser au maximum l'architecture pipeline des unités de calcul, il est intéressant de répéter plusieurs fois la même opération d'affiler. Cela est cependant difficile dans cette application à cause de la forte dépendance des données. Cependant, comme on peut le voir sur la figure ci-dessous, la micro-architecture `Sandy Bridge` dispose sur l'ordonnanceur de plusieurs ports menant à des unités de calcul SIMD, dont une pour les additions et une pour les multiplication, ce qui permet de limiter les cycles vides.

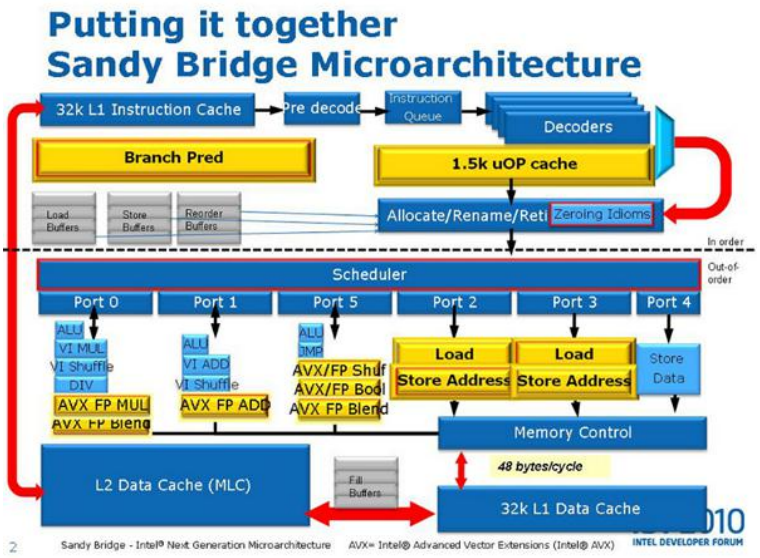


FIGURE 3.7 – Microarchitecture `Sandy Bridge`

Pour tout de même utiliser au maximum les unités de calculs il est possible de procéder à un calcul de type entrelacé.

C'est à dire réaliser la même opération plusieurs fois sur différentes données plutôt que de faire toutes les opérations une seule fois de manière séquentielle sur toutes les données. Cela permet de s'affranchir en partie de la dépendance entre données.

FIGURE 3.8 – Schématisation du calcul entrelacé

Comme on peut le voir sur la figure ci-dessous, cela permet d'améliorer une fois de plus les résultats, et de manière proche entre simple précision et double précision.

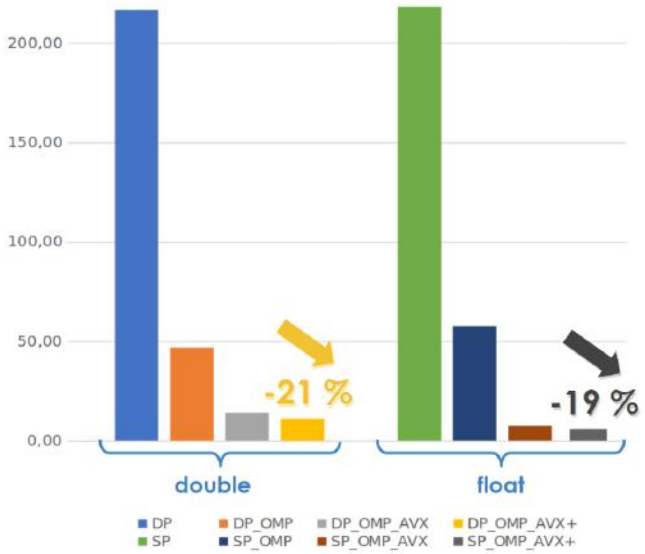


FIGURE 3.9 – Gain en temps d'exécution de l'utilisation des optimisations SIMD

3.1.2 Implémentation en virgule fixe

3.1.2.1 Généralités et mise en œuvre dans l'algorithme

Les données en virgule flottantes sont pratiques à utiliser mais, dans certains cas, l'utilisation d'opérateurs entier est plus avantageuse. Les opérations en virgules fixes peuvent être réalisées à partir des opérateurs entiers classiques.

De plusn les données en virgules fixe permettent une personnalisation du format des données. Dans notre cas une précision dans la partie décimale est requise, il est donc inutile d'avoir une partie entière trop importante :

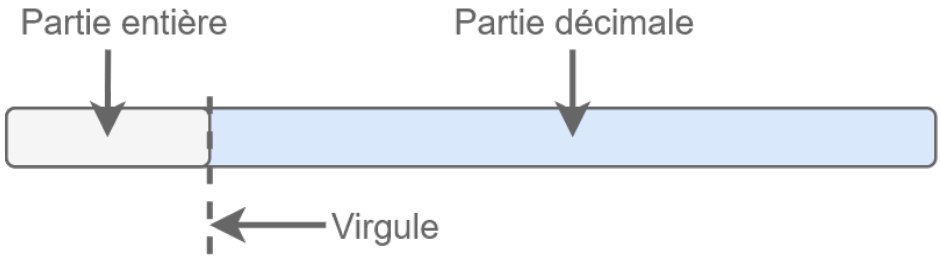


FIGURE 3.10 – Format d'une variable en virgule fixe de 32 bits

Le format ici adopté est `25/32`, c'est à dire qu'il y a 25 bits pour la partie décimale sur 32 bits au total. Ce format a été déterminé à l'aide d'une analyse de la plage de valeur des données utilisées et par des tests sur différents formats. Ce format permet de garder une partie décimale suffisante pour pouvoir zoomer assez loin (plus qu'en float) mais garde une partie entière suffisante pour éviter de voir apparaître des artefacts.

Une des difficultés apportées par le calcul en virgule fixe est que le résultat d'une multiplication de 32 bits retourne un résultat sur 64 bits il faut donc par la suite découper le résultat pour le ramener sur 32 bits. Pour se faire il faut restituer où est la virgule et récupérer un segment du bon format, le schéma ci-dessous illustre le processus :

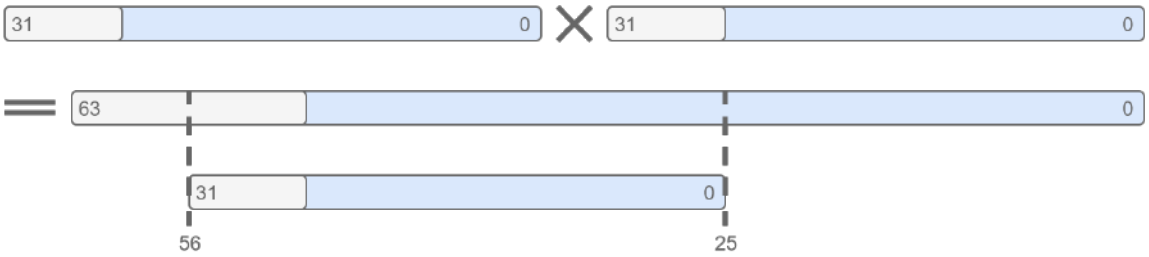


FIGURE 3.11 – Synthèse des résultats CPU obtenus sur un PC

Il est généralement nécessaire de faire une saturation au moment du découpage pour conserver des données cohérentes.

3.1.2.2 Parallélisme avec OpenMP

Comme précédemment, le programme en virgule fixe peut-être parallélisé avec OpenMP.

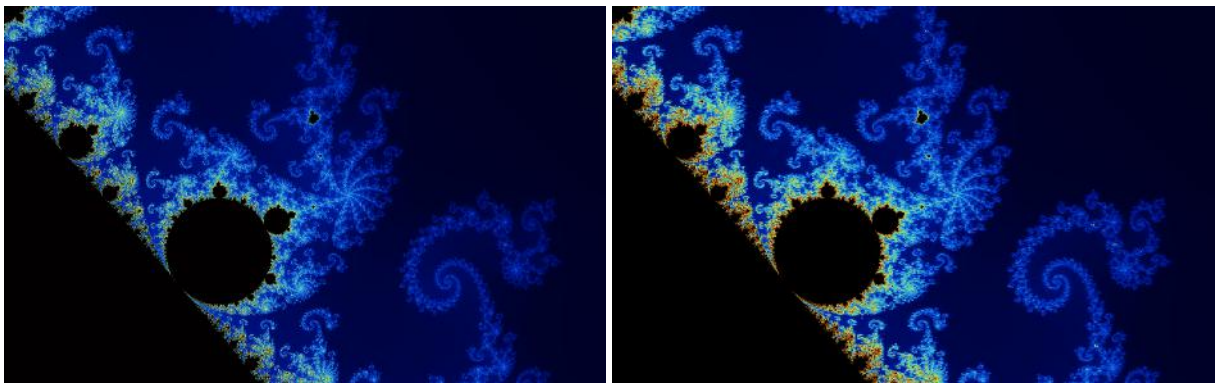
3.1.2.3 Parallélisme et vectorisation avec SSE

Le jeu d'instruction SSE permet d'offrir des intrinsèques SIMD manipulant des opérateurs entiers. Contrairement aux registres `AVX`, les registres `SSE` sont de 128 bits, ce qui veut dire que qu'on ne peut traiter que 4 données 32 bits à la fois.

On peut noter l'existence du jeu d'instruction `AVX2`, qui étant la plupart des instructions `SSE` sur des registres de 256 bits. Son utilisation aurait été intéressante mais demande d'avoir une micro-architecture Haswell ou plus récente pour Intel.

La principale difficulté du traitement de données à virgule fixe en SIMD sur SSE est de gérer les multiplications. D'autant plus que SSE manque de certaines intrinsèques qui auraient été utiles, comme `_mm_packs_epi64` par exemple, ce qui force à utiliser plusieurs opérations pour atteindre le même résultat.

Pour gagner en temps d'exécution, on peut choisir de ne pas implémenter de saturation pour les multiplications (qui aurait été gérée par `_mm_packs_epi64` si elle existait), mais cela amène à des artefacts dans l'image, comme on peut le voir sur la figure ci-dessous.



(a) Version à virgule fixe scalaire (b) Version à virgule fixe SIMD sans saturation

FIGURE 3.12 – Phénomène d'artefact en l'absence de saturation sur les multiplications

Il faut noter que l'utilisation de la virgule fixe en SIMD demande un temps de développement bien plus important que les version précédemment présentées.

3.2 Synthèse des résultats

Les simulations effectuées sur les modules de convergence précédemment développés, donnent sur les 8 scénarios de test les résultats suivants, en termes de temps médian.

Temps d'exécution médian									
N	Convergence	1	2	3	4	5	6	7	8
1	DP	44,47	131,10	348,46	283,24	14,17	216,63	44,72	119,92
2	DP_OMP	9,87	24,62	58,50	54,38	5,17	46,86	14,84	24,69
3	DP_OMP_AVX	3,11	7,53	17,96	15,54	2,02	13,97	4,52	7,27
4	DP_OMP_AVX+	2,47	6,12	15,18	12,60	1,61	11,02	3,51	5,75
5	DP_OMP_AVX++	3,75	9,79	22,41	19,87	2,31	17,10	5,44	9,04
6	SP	26,93	114,89	323,90	238,88	14,55	218,02	45,21	120,40
7	SP_OMP	6,89	27,60	66,42	57,73	6,29	57,68	18,15	30,62
8	SP_OMP_AVX	1,43	3,63	9,73	6,90	1,31	7,46	2,42	3,83
9	SP_OMP_AVX+	1,20	3,00	8,17	5,83	1,20	6,07	1,87	2,99
10	SP_OMP_AVX++								
11	FP	29,47	112,52	337,06	123,58	14,96	210,55	43,89	116,31
12	FP_OMP	7,85	25,45	68,31	89,90	5,62	53,83	15,42	28,34
13	FP_OMP_SSE	4,08	13,02	35,49	44,52	3,20	29,92	8,18	14,89

FIGURE 3.13 – Synthèse des résultats obtenus sur CPU uniquement pour une PC [4]

Les résultats grisés sont ceux qui sont jugés comme peu significatifs dans la mesure où une partie des calculs se font au delà de la limite de précision des formats.

On peut remarquer que malgré des performances alléchantes sur le papier, l'utilisation de la virgule fixe sur CPU reste moins performante que la version à virgule flottante en double précision, qui traite le même nombre de données en parallèle. La principale hypothèse pour ces résultats est que les opérations en virgule fixe, bien qu'utilisant des opérateurs plus rapide, demandent des étapes intermédiaires coûteuses en cycles d'horloges.

4. Temps d'exécution médian obtenu pour 10 mesures successives,
sur Ubuntu 18.04.3 LTS 64 bits avec un CPU Intel® Core™ i7-3610QM @ 2.30GHz x 8.
Programme compilé avec clang 6.0.0-1ubuntu2 avec le flag -O3

3.3 Programmation sur GPU

Après avoir programmé un premier programme en C réalisant la fractale de Mandelbrot de manière non optimisée, une autre optimisation a consisté en l'utilisation du GPU (Graphic Processor Unit) pour accélérer les calculs.

En effet, Nvidia, un développeur de processeur graphique, vante la vitesse des GPU pour l'exécution des algorithmes parallélisables. Comme l'algorithme réalisant la fractal de Mandelbrot est justement parallélisable, c'est une belle opportunité pour vérifier ces dires.
De plus, un autre projet réalisé durant cette présente année scolaire (le projet **Galaxeirb**), a permis de prendre en main l'utilisation de la bibliothèque CUDA pour pouvoir programmer de manière efficace sur les GPU de Nvidia.

Des protocoles de tests ont été mis en place permettant une comparaison objective des performances de la solution optimisée sur CPU et celle utilisant le GPU. Différents critères seront comparés, tels que :

— la vitesse d'exécution du programme
— la vitesse de développement du programme
— la consommation électrique
— le coût budgétaire

3.3.1 La bibliothèque CUDA

Le **GPU** (Graphic Processor Unit) est processeur optimisé pour réaliser plusieurs calculs en parallèles. En effet, il possède des centaines voir des milliers de coeurs de calculs (640 CUDA Cores pour la GTX 1050, 2560 CUDA Cores pour la GTX 1080) spécialisés dans le calculs à virgules flottantes sur 32 bits. Cette spécificité lui permet de réaliser de manière très rapide les calculs graphiques (traitement d'image, affichage d'un environnement 3D, ...).

Pour réussir à tirer parti de ces coeurs de calculs, la bibliothèque **CUDA** est utilisée. Elle permet d'utiliser les GPU récents de la marque Nvidia.

Avec cette bibliothèque, une fois que le programme en C est développé, il est aisé "d'envoyer" les calculs sur le GPU. Pour cela, 2 types de fichiers de programmation sont utilisée :

— le "**.cpp**" (avec le ".hpp" associé) qui décrit le programme qui sera exécuté sur le CPU.
— le "**.cu**" (avec le ".cuh" associé) qui décrit le programme qui sera exécuté sur le GPU.

Ensuite, les fonctions supplémentaires suivantes sont ajoutées au programme :

— L'allocation dynamique d'une variable "*host*" et d'une variable "*device*" est nécessaire. La variable "*host*" est stockée dans la RAM du CPU et la variable "*device*" est l'image de cette variable stockée dans la mémoire RAM du GPU. Pour cela, la variable "*device*" est créée en RAM du CPU avec **Malloc()** (tout comme la variable "*host*") et "envoyée" sur la RAM du GPU avec la fonction **CudaMalloc()**.

— Ensuite, si une donnée déjà existante doit être traitée par le GPU, il est possible de l'envoyer en RAM du GPU avec la fonction **CudaMemCpy()**, avec l'argument **HostToDevice**. Cela n'est pas utile pour ce programme de fractale.

— Ensuite, avec les données présentes sur la RAM du GPU, il est alors possible d'appeler la fonction de calcul dont la description se situe dans le fichier "**.cu**". Il s'agit de la fonction "**compute_cuda()**".

— Ensuite, pour récupérer le résultat du calcul du GPU, il faut réaliser un tranfert de la RAM du GPU vers la RAM du CPU avec la fonction **CudaMemCpy()**, avec l'argument **DeviceToHost**. Dans le cas de la fractale de Mandelbrot, la donnée renvoyée est un tableau contenant les itérations pour chacune des positions calculées. Le processeur peut alors afficher à l'écran le résultat en associant chaque nombre d'itérations à une couleur.

— Enfin, pour libérer la mémoire des 2 RAM, il faut utiliser **free()** et **cudaFree()**.

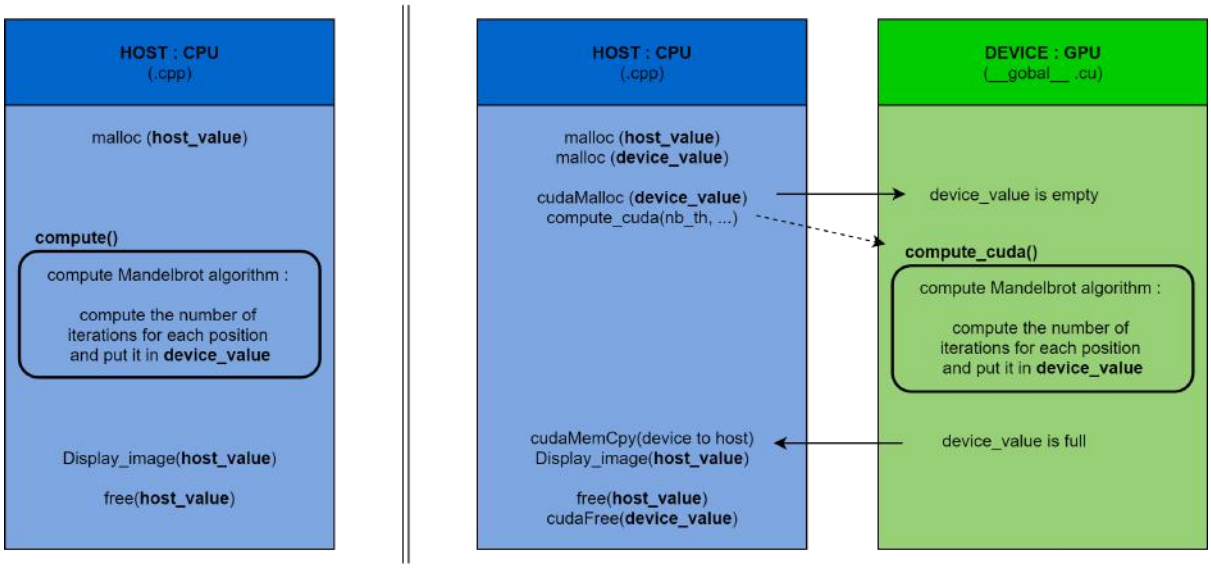


FIGURE 3.14 – Principales fonctions des programmes qui s'exécutent sur le CPU seulement (à gauche) et sur le CPU et le GPU (à droite)

3.3.2 La théorie sur le fonctionnement des GPU Nvidia

Après quelques recherches sur le fonctionnement des GPU, voici un résumé de ce qui a pu être trouvé. Il faut cependant garder en tête qu'il peut s'agir d'approximation : le constructeur ne divulguant pas toutes les explications du fonctionnement de ses produits, les explications trouvées sur les forums semblent parfois être liée à une interprétation des utilisateurs.

Depuis 2007, la plupart des GPU de chez Nvidia sont construits autour d'une architecture CUDA, c'est à dire basée sur une architecture évolutive. Cela signifie qu'il existe des paramètres à ajuster qui permettent à un même programme d'être exécuté différemment selon la configuration choisie.

Évidemment, il ne s'agit pas d'un FPGA : l'architecture matérielle ne changera jamais. Mais les ressources matériels seront utilisées de différentes manières selon ce paramétrage.

L'architecture matérielle des GPU est hiérarchisée sous différentes couches :

— Elle est composée de plusieurs unités de calculs (entre 100 et 10000 aujourd'hui), nommés **Streaming Processors** (**SPs**) (ce sont les fameux Cuda Cores des GPU de Nvidia). Ces processeur sont relativement simples comparés aux coeurs de calculs des CPU : ils réalisent les calculs en continu (d'ou le nom "streaming") sans réaliser de prédictions. De plus, ils sont spécialisés dans le calcul à virgule flottante à simple précision (32 bits) mais aussi parfois sur les entiers de 32 bits (voir **Figure 3.16**, pour une Compute Cability de 6.1 pour la GTX 1050).

— Les SPs sont rassemblés par groupes de même nombre dans des **Streaming Multiprocessors** (SMs). Chaque SM a donc un certain nombre fixe de SPs. Par exemple, la GTX 1080 Ti possède 128 SPs/SM (Tableau 1 de NVIDIA TURING GPU ARCHITECTURE). Les SMs sont également composées d'ordonnanceurs, d'un cache et d'une mémoire partagée entre les SPs.

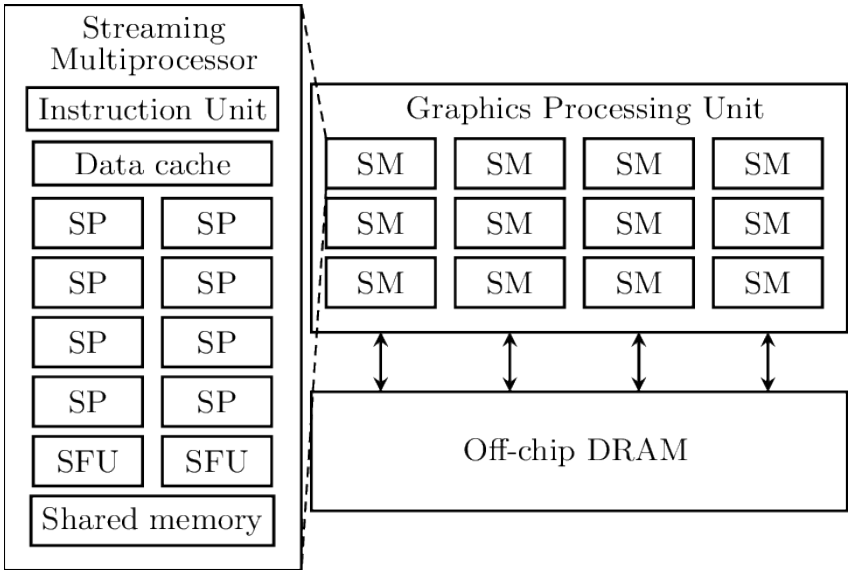


FIGURE 3.15 – Schéma de l'architecture matérielle CUDA d'un GPU
Source : Simplified schematic of a GPU with 96 SPs

Avec cette architecture matérielle, le GPU exécute **les instructions** d'une certaine manière :

— Un **thread** (un processus) est un ensemble d'instructions qui peuvent être exécutée par un SP.

— Chaque SM prend en entrée un ensemble composé de 32 threads nommé **Warp** (voir le chapitre 4 du NVIDIA CUDA C Programming Guide).
Ainsi on peut faire les approximations suivantes : si un GPU possède 8 SPs/SM, il traitera 1 Warp (32 threads) en 4 fois par SM, soit en exécutant 8 threads par SM en parallèle. Si ce même GPU a 5 SM, il exécutera 40 threads en parallèle. Si un autre GPU possède 128 SPs/SM, il peut traiter 4 Warp en 1 fois, en parallèle, par SM.

	Compute Capability							
	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x
16-bit floating-point add, multiply, multiply-add	N/A	N/A	N/A	256	128	2	256	128
32-bit floating-point add, multiply, multiply-add	192	192	128	128	64	128	128	64
64-bit floating-point add, multiply, multiply-add	8	64[1]	4	4	32	4	4	32[1]
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base 2 exponential (exp2f), sine (__sinf), cosine (__cosf)	32	32	32	32	16	32	32	16
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	160	128	128	64	128	128	64
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	32	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	64[3]
24-bit integer multiply (__[u]mul24)	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.
32-bit integer shift	32	64[2]	64	64	32	64	64	64
compare, minimum, maximum	160	160	64	64	32	64	64	64
32-bit integer bit reverse, bit field extract/insert	32	32	64	64	32	64	64	Multiple instruct.
32-bit bitwise AND, OR, XOR	160	160	128	128	64	128	128	64
count of leading zeros, most significant non-sign bit	32	32	32	32	16	32	32	16
population count	32	32	32	32	16	32	32	16
warp shuffle	32	32	32	32	32	32	32	32[3]
sum of absolute difference	32	32	64	64	32	64	64	64
SIMD video instructions vabsdiff2	160	160	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.
SIMD video instructions vabsdiff4	160	160	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	64
All other SIMD video instructions	32	32	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.
Type conversions from 8-bit and 16-bit integer to 32-bit types	128	128	32	32	16	32	32	16
Type conversions from and to 64-bit types	8	32[2]	4	4	16	4	4	16[3]
All other type conversions	32	32	32	32	16	32	32	16

FIGURE 3.16 – Tableau des débits des instructions arithmétiques native des GPU de Nvidia en fonction de "Compute Capability" (en Nombre de résultats par cycle d'horloge par SM)

Ensuite, pour pouvoir utiliser au mieux cette architecture hiérarchisée, il faut également hiérarchiser les instructions envoyées par l'hôte vers le GPU. Pour cela, le compilateur NVCC de la bibliothèque CUDA (qui permet de compiler un exécutable sur une plate-forme possédant une architecture CUDA) demande 3 paramètres au programmeur. Ces paramètres sont évidemment dépendant de l'architecture, mais il n'est pas nécessaire de comprendre son fonctionnement pour bien paramétrer son programme. Ainsi, ces 3 paramètres n'ont pas de lien direct avec la structure de l'architecture du GPU.

Les 3 paramètres que le **programmeur** doit choisir sont les suivants :

— **Le nombre de threads par blocs**. Du coté de la compilation par l'hôte (le CPU), les threads sont regroupés par blocs. Avec ce paramètre, on indique combien de threads on veut mettre dans chaque bloc.

Comme vu précédemment, ce qui va rentrer dans les SMs, ce sont les Warps composés de 32 "places" pour y mettre des threads. Les Warps sont remplis en fonctions du remplissage des blocs. Si un bloc a 48 threads, alors 2 Warps seront utilisés, mais ne seront pas plein car 64 places sont disponibles. Pour optimiser les calculs il faut remplir les Warp au maximum.

⇒ Pour cela, il faut donc que le nombre de thread par bloc soit un multiple de 32. Il ne doit cependant pas excéder 1024 (actuellement, pour les GPU récents, dont la GTX 1050).

— **La dimension de la grille**. La grille représente "tout le GPU". Il faut alors indiquer si l'on veut coder en 1D, 2D ou 3D. Il faut ensuite indiquer en combien de bloc cette grille peut être découpée.

— **le nombre de blocs**. Il s'agit du nombre de blocs que l'on veut utiliser. Ce paramètre est dépendant de la donnée à traiter et du nombre de threads par bloc. Toutes les threads d'un même bloc auront accès à la même mémoire partagée (voir **Figure 3.17**). Pour déterminer le nombre de bloc à utiliser on utilise la formule suivante :

$$NbBlock = \frac{TailleDonnee + NbThreadsParBloc - 1}{NbThreadsParBloc}$$

Dans le cas où les données traitées sont en plusieurs dimensions (2D et 3D), il faut indiquer le nombre de blocs et le nombre de threads utilisés dans chacune des dimensions.

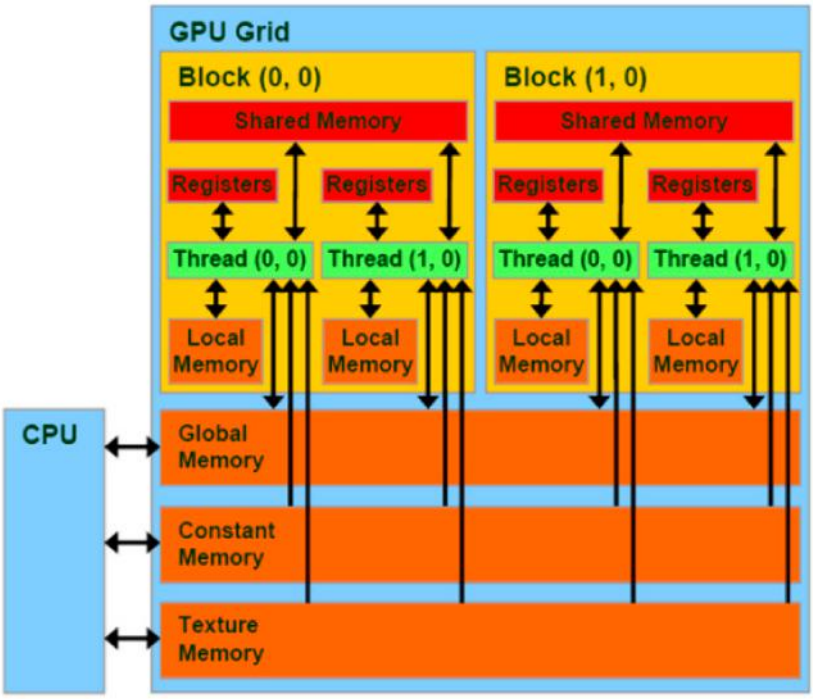


FIGURE 3.17 – GPU hiérarchisé pour le programmeur

Ci-dessous, voici un exemple de la manière de paramétrer un GPU avec CUDA en 2D. La fonction `compute()` est appelée par le CPU dans un ".cpp" et décrite dans un fichier ".cu". Cette fonction paramètre le GPU. Elle appelle alors une autre fonction `kernel_compute()` dont le contenu est exécuté sur le GPU.

Exemple de la manière de paramétrer un GPU en 2D

```
__global__ void kernel_compute(double zoom, double offsetX, double
↪  offsetY, unsigned int max_iters, int IMAGE_WIDTH, int
↪  IMAGE_HEIGHT, unsigned short *device_value) {

    [...]
}

void compute (int nthreadsX, int nthreadsY, double zoom, double
↪  offsetX, double offsetY, unsigned int max_iters, int
↪  IMAGE_WIDTH, int IMAGE_HEIGHT, unsigned short *device_value) {

  int numBlocksX = ( IMAGE_WIDTH + ( nthreadsX - 1 ) ) / nthreadsX;
  int numBlocksY = ( IMAGE_HEIGHT + ( nthreadsY - 1 ) ) / nthreadsY;

  dim3 DimBlock(nthreadsX, nthreadsY, 1);
  dim3 DimGrid(numBlocksX, numBlocksY, 1);

  kernel_compute<<<DimGrid, DimBlock>>>( zoom, offsetX, offsetY,
   ↪  max_iters, IMAGE_WIDTH, IMAGE_HEIGHT, device_value);

}
```

3.3.3 Le choix du type de variable utilisée

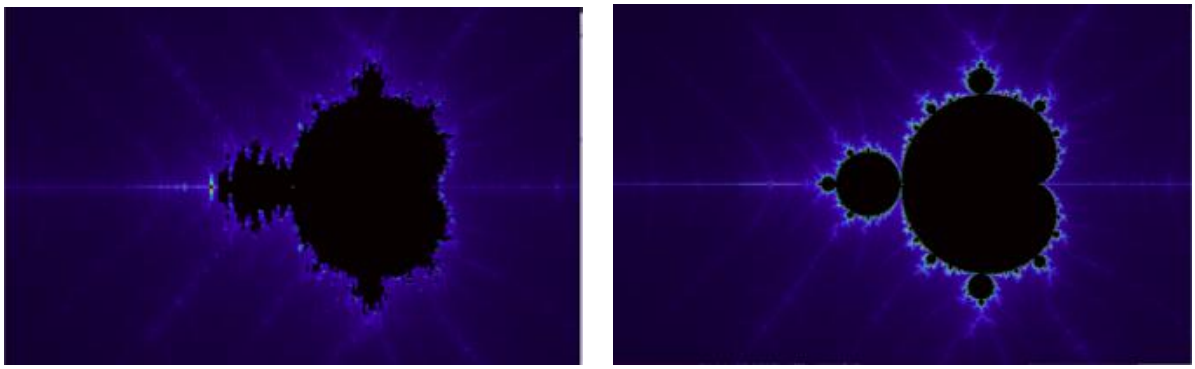
Le GPU étant spécialisé dans le calcul à virgule flottante, il a été décidé de ne tester que 2 types de variables pour les données traitées :

— Les **Floats**, ou flottant à simple précision, c'est à dire codés sur 32 bits.

— Les **Doubles**, (flottant à double précision) codés sur 64 bits.

La virgule fixe n'a pas été abordée car le calcul d'entiers n'est pas plus performant que le calcul flottant sur les GPU.

Les Floats permettent, en théorie, de réaliser des calculs 32 fois plus vite qu'avec des doubles avec une GTX 1050 (voir **Figure 3.16**). Cependant, les approximations de calculs se font rapidement ressentir quand on zoome dans la fractale avec les Floats (voir **Figure 3.18**). C'est pourquoi, la plupart des optimisations ont été testées avec le type Double.



(a) Float (b) Double

FIGURE 3.18 – Images générées, en fonction du type des variable, en zoomant dans la fractale avec les paramètres suivants :
`offsetX = -1.7864343; offsetY = 0.0; zoom = 2.0e-08; max_iters = 4096; IMAGE_WIDTH = 1280; IMAGE_HEIGHT = 800;`

3.3.4 Optimisation 1 : calculs du meilleur nombre de threads par blocs

Selon la théorie expliquée dans la **partie 3.3.2**, on retient que la manière dont va être exécuté de le programme sur le CPU dépend de 3 paramètres qui sont :

— La dimension de la grille : ici c'est une grille en 2D

— Le nombre de blocs sur la grille

— Le nombre de threads par bloc : à choisir

En réalité, ces 3 paramètres sont liés les uns aux autres et à la taille de la donnée traitée. C'est pourquoi, le seul paramètre que l'utilisateur a à choisir est le nombre de threads par bloc.

Pour choisir quel est la valeur de ce paramètre qui permet au programme d'être exécuté le plus rapidement, plusieurs valeurs sont testées. Pour simplifier le problème, seuls les valeurs de threads par blocs ayant une racine carrée entière seront testées. Cela correspond aux valeurs `nbthreads` suivantes :

$$\texttt{nthreads}^2 = \texttt{nthreadsX} \times \texttt{nthreadsY}$$

avec

$$\texttt{nthreadsX} = \texttt{nthreadsY}$$

ce qui donne 32 valeurs possibles, car la valeur maximale du nombre de threads par bloc est de 1024 et $\sqrt{1024} = 32$.

On retient également de la théorie sur les GPU Nvidia, que le nombre de threads par blocs doit être un multiple de 32 pour être optimisé. Ainsi, on s'attend à voir une baisse du temps d'exécution pour : 64, 256, 576, 1024 threads par blocs.

Pour réaliser les mesures, 8 tests différents ont été répétés chacun 10 fois et la valeur médiane des 10 répétitions est enregistrée. Pour pouvoir comparer les tests entre eux, une normalisation est réalisée : pour chaque test, chaque mesure de médiane enregistrée

est divisée par la valeur de la médiane de la 8^e valeur (celle pour laquelle le nombre de threads vaut 64). Ainsi, le temps normalisé de la 8^e valeur de tous les tests vaut toujours 1. Enfin, une moyenne de chaque point est réalisé.

Le résultat des mesures est présenté en **Figure 3.19**.

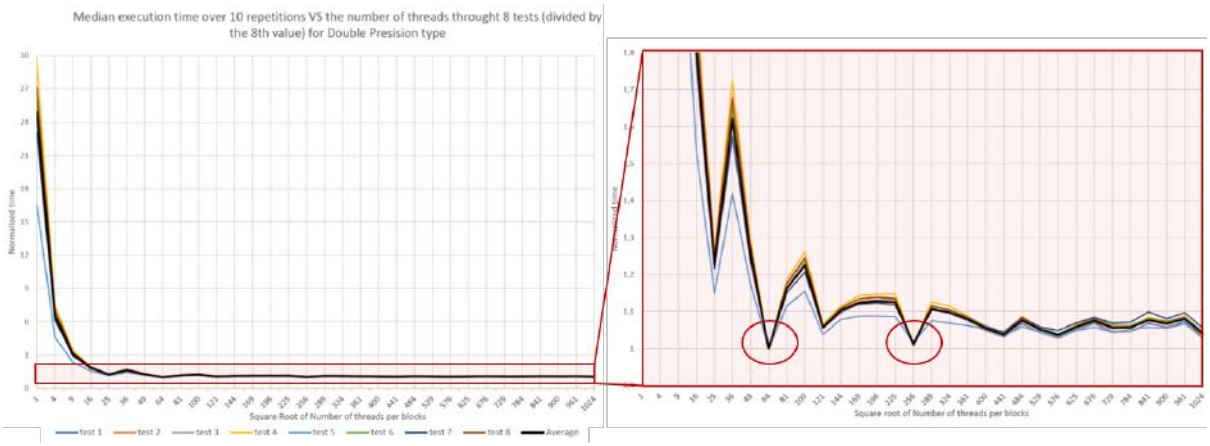


FIGURE 3.19 – Calcul du temps d'exécution médian sur 10 répétition sur 8 tests différents en fonction du nombre de threads par bloc (temps normalisé)

Premièrement, on remarque une grande amélioration de la rapidité à la suite de l'augmentation du nombre de threads par blocs, pour les 5 premières valeurs 1, 4, 9, 16 et 25 threads par blocs. En effet, le programme est 20 fois plus rapide pour 25 threads par bloc que pour 1. On peut donc comprendre qu'en dessous de 25 threads par bloc, **tout le GPU n'est pas utilisé pour paralléliser les calculs.**

Deuxièmement, à partir de la 8^e valeur (64), on peut penser que tout le GPU est sollicité puisque l'influence du nombre de threads par blocs n'a plus beaucoup d'incidence sur la rapidité. Cependant, en regardant plus précisément, on remarque en effet un temps d'exécution plus court pour tous les tests ayant pour nombre de threads par bloc un multiple de 32. **C'est d'ailleurs 64 et 256 threads par blocs qui permettent la meilleure rapidité, c'est pourquoi ce sont ces nombres qui seront utilisés par la suite**.

3.3.5 Optimisation 2 : utilisation de double2

CUDA possède ses propres types de variables qui permettent un traitement et un transfert plus efficace des données. En effet, possédant une largeur de bus mémoire de 128 bits, la GTX 1050 est plus efficace si elle transfert des données qui font déjà 128 bits. C'est pour cela que le type **double2** (qui contient 2 doubles de 64bits) va être utilisé. Pour les floats, il existe les **float4** qui contiennent 4 floats de 32 bits. On notera que l'utilisation des float4, dans le cadre du projet **Glaxeirb**, ont permis de diviser les temps d'exécution par 3 à 4.

Etant donné que la fractale de Mandelbrot manipule des données complexe, il a été décidé d'utiliser les double2 ce cette façon :

— les 64 premiers bits du double2 sont consacrés à stocker la valeur **réelle** du nombre complexe manipulé

— les 64 derniers bits du double2 sont consacrés à stocker sa valeur **imaginaire**

Ainsi, l'algorithme qui était auparavant le suivant :

Exemple de la manière de paramétrer un GPU en 2D

```
double zReal = startReal;
double zImag = startImag;

for (unsigned short counter = 0; counter < max_iters; counter++) {
  double r2 = zReal * zReal;
  double i2 = zImag * zImag;
  zImag = 2.0f * zReal * zImag + startImag;
  zReal = r2 - i2 + startReal;
  if ( (r2 + i2) > 4.0f) {
    return counter;
  }
}
return max_iters - 1;
```

Devient ceci :

Exemple de la manière de paramétrer un GPU en 2D

```
double2 startZ = startValue;
double2 z1;
z1.x = startZ.x;
z1.y = startZ.y;

for (unsigned short counter = 0; counter < max_iters; counter++) {
  double2 z2;
  z2.x = z1.x * z1.x;
  z2.y = z1.y * z1.y;

  z1.y = 2.0f * z1.x * z1.y + startZ.y;
  z1.x = z2.x - z2.y + startZ.x;
  if ( (z2.x + z2.y) > 4.0f) {
    return counter;
  }
}
return max_iters - 1;
```

Les tests avec et sans double2 sont réalisés avec 256 threads par bloc. Les 2 tests présentent évidemment des temps d'exécution différents, mais **la différence est bien trop faible pour être différenciée du hasard**.

3.3.6 Optimisation 3 : fusion de fonctions

Le fichier "kernel.cu" qui contient le programme qui est exécuté sur le GPU possède les fonctions suivantes :

— `compute_cuda()` : c'est la fonction de paramètrage du GPU. Elle appelle ensuite la fonction `kernel_compute_cuda()`

— `kernel_compute_cuda()` : elle contient le code qui est exécuté sur le GPU et réparti les différentes taches sur les threads et blocs. Elle appelle la fonction `process_cuda()`

— `process_cuda()` : c'est la fonction qui contient l'algorithme de la fractale de Mandelbrot.

Dans l'optique de clarifier le programme et parce qu'il y avait un espoir que les appels de fonctions prennent du temps GPU, il a été décidé de copier le contenu de la fonction `process_cuda()` dans `kernel_compute_cuda()` et d'effacer `process_cuda()`.

L'amélioration (ou le ralentissement) est malheureusement trop faible pour être discerné du hasard.

3.3.7 Optimisation 4 : utilisation de la Shared Memory

Cette optimisation vise à faire une copie de ce qui se situe en mémoire RAM du GPU, dans une autre mémoire beaucoup plus rapide : la Mémoire Partagée, ou Shared Memory.

Cette mémoire est beaucoup plus proche des SPs (voir **Figures 3.15** et **3.17**) et beaucoup plus rapide (en écriture et en lecture). Ainsi, si la donnée est sollicitée plusieurs fois dans le calcul qui suit, il vaut mieux perdre un peu de temps à réaliser l'instruction de copie (copie de la RAM vers la Mémoire Partagée, puis copie de la Mémoire Partagée à la RAM une fois le calcul terminé), pour en gagner à chaque nouvel accès à la variable durant les calculs.

Réaliser cette optimisation n'a pas été possible car elle nécessite l'utilisation de tableau, c'est-à-dire d'une grande quantité de donnée en provenance de la RAM du CPU. Cependant aucune donnée à traiter n'est envoyée depuis le CPU vers le GPU. Toutes les données sont directement créées sur le GPU, et ces données sont petites, ce qui signifie qu'elle sont sûrement stockées dans la **mémoire cache**. La mémoire cache étant la mémoire la plus rapide du GPU, **il n'est pas possible d'améliorer les performances en utilisant la mémoire partagée**.

3.3.8 Optimisation 5 : utilisation de la Pinned Memory

Cette optimisation consiste à essayer d'optimiser les transferts de données en les réalisant de manière asynchrone. Ainsi, lorsqu'une donnée est traitée, elle est directement renvoyée vers le CPU sans attendre le résultat suivant. Sans utilisation de Pinned Memory, le GPU attend que tout une suite d'instruction soit terminée pour envoyer ensuite le résultat au CPU.

Lors du projet Galaxeirb, l'utilisation du Pinned Memory a permis de diviser les temps de calculs par 2 à 3 selon les groupes. Cependant, tout comme pour l'optimisation avec la mémoire partagée, cette optimisation n'est applicable que lorsqu'il y a un grand nombre de données à traiter venant du CPU. Ici, aucune donnée ne provient du CPU et les données sont créées dans le cache. **Il est donc normal de ne pas observer d'amélioration**.

3.3.9 Optimisation 6 : utilisation d'intrinsèques de calcul

À l'image de ce qui a été réalisé avec AVX et SSE pour le CPU, il est possible d'utiliser des intrinsèques de calculs avec CUDA. Cette optimisation consiste à utiliser des fonctions de la bibliothèque CUDA pour réaliser les calculs (addition, multiplication, division, ...) de manière optimisée pour le GPU.

On trouve l'ensemble des intrinsèques sur le lien suivant : CUDA Double Presision Intrinsics.

Voici un programme avec des opérateurs classiques :

Programme sans intrinsèques

```
z2.x = z1.x * z1.x;
z2.y = z1.y * z1.y;

z1.y = 2.0f * z1.x * z1.y + z.y;
z1.x = z2.x - z2.y + z.x;
```

Et voici le même programme codé avec les intrinsèques :

Programme avec intrinsèques

```
z2.x = __dmul_rn( z1.x, z1.x);
z2.y = __dmul_rn( z1.y, z1.y);

z1.y = __fma_rn(__dmul_rn(2.0f, z1.x), z1.y, z.y);
z1.x = __dadd_rn(__dsub_rn(z2.x, z2.y), z.x);
```

Cependant, dans ce cas précis, l'utilisation d'intrinsèques **ralenti** l'exécution du programme. On peut donc en conclure que le compilateur optimise déjà le programme d'une meilleure manière que ce qui est possible de réaliser manuellement.

3.3.10 Optimisation 7 : utilisation du type Float

Toutes les optimisations actuelles ont été réalisées en utilisant des flottants à double précision. Cependant, comme présenté en **partie 3.3.3**, le GPU GTX 1050 est bien plus optimisé pour réaliser des calculs sur flottant à simple précision. En effet, il serait 32 fois plus rapide avec ces derniers.

Pour pouvoir mesurer la différence entre le type Float et le type Double, seuls les tests 5, 6, 7 et 8 sont utilisés car ce sont des tests qui ne "zooment" pas trop dans la fractale pour que les approximations de calculs ne soit pas perceptibles. Encore une fois ce sont les temps médians sur 10 répétitions de chaque test qui sont relevés.

Les résultats sont présentés **Figure 3.20**.

Median execution time (s) VS SP and DP				
Test Number	5	6	7	8
SP GPU	0,753	1,681	0,621	0,87
DP GPU	1,55	13,98	3,38	7,47
ratio DP/SP	2,06	8,32	5,44	8,59
GPU solution	Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4 CPU + Nvidia GTX 1050 2GB			

FIGURE 3.20 – Calcul du temps d'exécution entre l'utilisation de float (simple précision) et double (double précision) sur 4 tests

On remarque alors que :

— comme attendu, le rapport du temps d'exécution en utilisant des floats est plus court

— comme attendu, il est plus que 2 fois plus cours, sauf pour le test 5

— contrairement à ce qui était attendu, le rapport n'est pas de 32, mais compris entre 2 et 9 selon les tests.

On remarquera que les rapports entre temps d'exécution avec float et avec double ne sont pas liés au hasard : même en changeant le nombre de threads par bloc d'un même test, ce rapport reste sensiblement le même (pour les nombres de threads par bloc au dessus de 16) (voir **Figure 3.21**) .

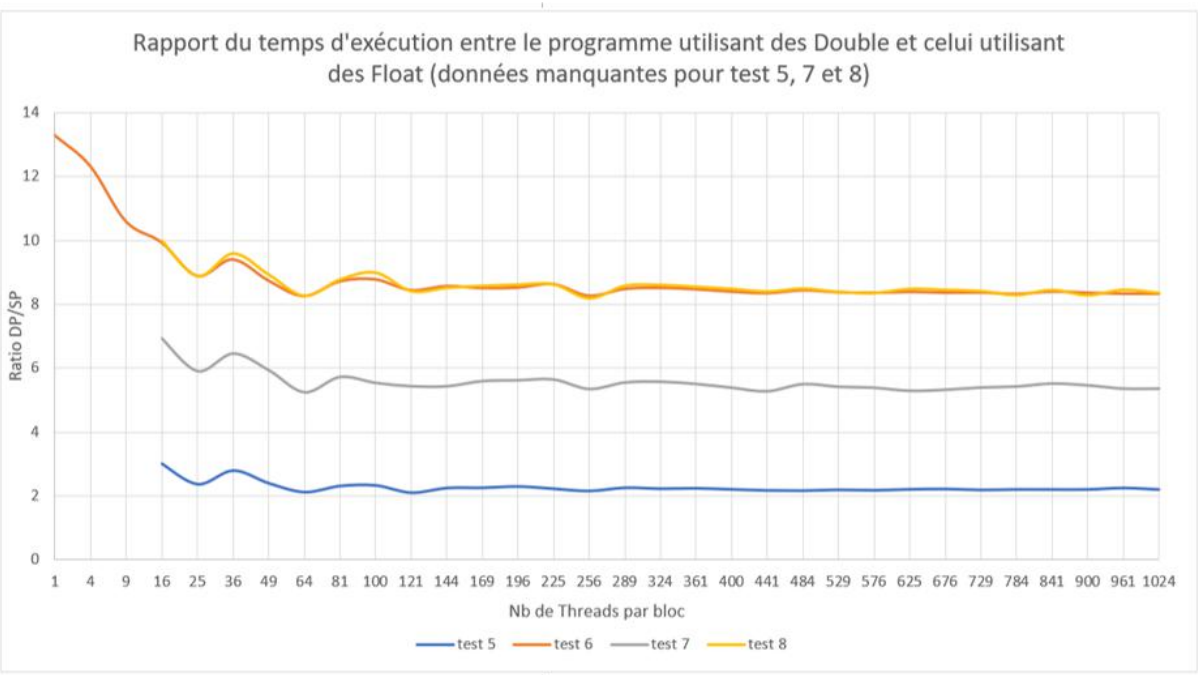


FIGURE 3.21 – Calcul du temps d'exécution entre l'utilisation de float (simple précision) et double (double précision) sur 4 tests en fonction du nombre de threads par bloc

La raison pour laquelle le rapport n'est pas de 32 reste inconnue. C'est peut-être parce qu'avec les float, le temps de transfert des données n'est plus négligeable devant le temps des calculs. En effet, les simulations lentes (celles pour lesquelles il y a plus de calculs)

sont plus "boostée" par l'utilisation de float que les simulations rapides. Cependant cela n'a pas pu être vérifié, puisque le logiciel Nvidia Profiler n'a jamais fonctionné.

3.3.11 Résultats et conclusion

Pour conclure, on peut dire que sur les 6 optimisations essayée en utilisant le type double, 5 d'entre elles n'ont pas améliorée les performances. De plus, la solution GPU, aussi alléchante semblait-elle (causée par le marketing de Nvidia), n'est pas forcement la plus idéale.

En effet, en utilisant les flottants à double précision, le CPU arrive à exécuter le programme plus rapidement que la solution utilisant le GPU sur 6 des 8 tests.

Median execution time (s) using DP								
Test Number	1	2	3	4	5	6	7	8
DP GPU	2,89	8,55	21,91	16,06	1,55	13,98	3,38	7,47
DP CPU AVX+	2,47	6,12	15,18	12,6	1,61	11,02	3,51	5,75
ratio GPU/CPU	0,85	0,72	0,69	0,78	1,04	0,79	1,04	0,77
CPU solution	Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8 CPU							
GPU solution	Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4 CPU + Nvidia GTX 1050 2GB							

FIGURE 3.22 – Calcul du temps d'exécution de la meilleure solution sur GPU et sur CPU

De plus théoriquement, la solution utilisant le GPU consomme environ 3 fois plus que celle utilisant le CPU. En effet, pour faire fonctionner le GPU de 75W, il faut également utiliser le CPU de 45W.

Cependant, il faut garder en tête que ces consommations sont théoriques. Un autre groupe (celui réalisant du Deep Learning) ayant réalisé un test de consommation mesurait des puissances de 27W (pour le GPU seulement) en plein fonctionnement pour une GTX 1060 ayant une puissance théorique de 120W. Il faudrait mesurer précisément la consommation de chaque solution pour en tirer une réelle conclusion.

Power consumption	
CPU	45W
CPU+GPU	120W
FPGA	<1W
CPU solution	Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8 CPU
GPU solution	Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4 CPU + Nvidia GTX 1050 2GB

FIGURE 3.23 – Puissance théorique consommée par chaque solution

Enfin, en utilisant le type Float (pour le CPU et le GPU), le GPU se revèle cette fois-ci le plus rapide, d'un facteur 1,59 à 3,6. On notera que pour les tests 6, 7 et 8, si la puissance consommée par le CPU est 3 fois plus faible, il est aussi 3 fois plus lent. Ce qui donne un rapport vitesse/puissance identique pour les 2 solutions.

Median execution time (s) using SP				
Test Number	5	6	7	8
SP GPU	0,753	1,681	0,621	0,87
SP CPU AVX+	1,2	6,07	1,87	2,99
ratio GPU/CPU	1,59	3,61	3,01	3,44
CPU solution	Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8 CPU			
GPU solution	Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4 CPU + Nvidia GTX 1050 2GB			

FIGURE 3.24 – Calcul du temps d'exécution de la meilleure solution sur GPU et sur CPU

3.4 Programme d'affichage

3.4.1 Affichage et entrées utilisateur

Le programme d'affichage utilise la bibliothèque SFML, basée sur OpenGL. La partie affichage utilise principalement la méthode `setPixel` de la classe `sf::Image`. Les déplacements dans la fractale peuvent ce faire au moyen des touches du clavier :

- **Flèches directionnelles** : déplacements
- **P** : augmentation du nombre max d'itérations
- **O** : diminution du nombre max d'itérations
- **A** : zoom
- **Z** : dézoom

Cette partie a été fournie au début du projet. Le programme a cependant été enrichi par de nouvelles fonctionnalités :

- **F12** : capture d'écran
- **I** : affichage d'informations dans le terminal (position, zoom)
- **C** : activer/desactiver le point central
- **R** : re-calcul de l'image en cours

Afin de rendre les déplacements plus agréables dans la fractale, une fonctionnalité de déplacement avec la souris a été développée.

Ainsi, il est possible de zoomer directement sur un point donné en le pointant et en utilisant la molette. Il est également possible de se centrer sur un point en se déplaçant dessus. Ces fonctionnalités utilisent une transposition de l'espace de l'écran, dans lequel se trouve le pointeur de la souris, vers l'espace de Mandelbrot en fonction des paramètres actuels tels que les offsets et le zoom.

3.4.2 ColorMap

En plus du colormap de base, d'autres ont été crées. En effet, plusieurs approches de coloration de la fractale ont été testées.

La plus simple à implémenter est celle en nuances de gris, puisqu'il suffit de réaliser une association linéaire du nombre d'itérations avant convergence à un niveau de gris.

Un mode de couleur souvent utilisé et simple à mettre en place utilise l'espace de couleur HSV. Dans ce mode de couleur, on peut garder le paramètre "S" (saturation) et "V" (value) à une valeur constante pour chaque itération et faire varier "H" (hue) en fonction du nombre d'itération pour avoir des couleurs différentes.

Des variantes du mode de couleur de base et du mode de couleur HSV ont été crées. L'idée originale de ces modes "Loop" était de ne pas avoir de variation dans les couleurs déjà affichées si l'on augmente les itérations. Elle se base donc sur des modulos.

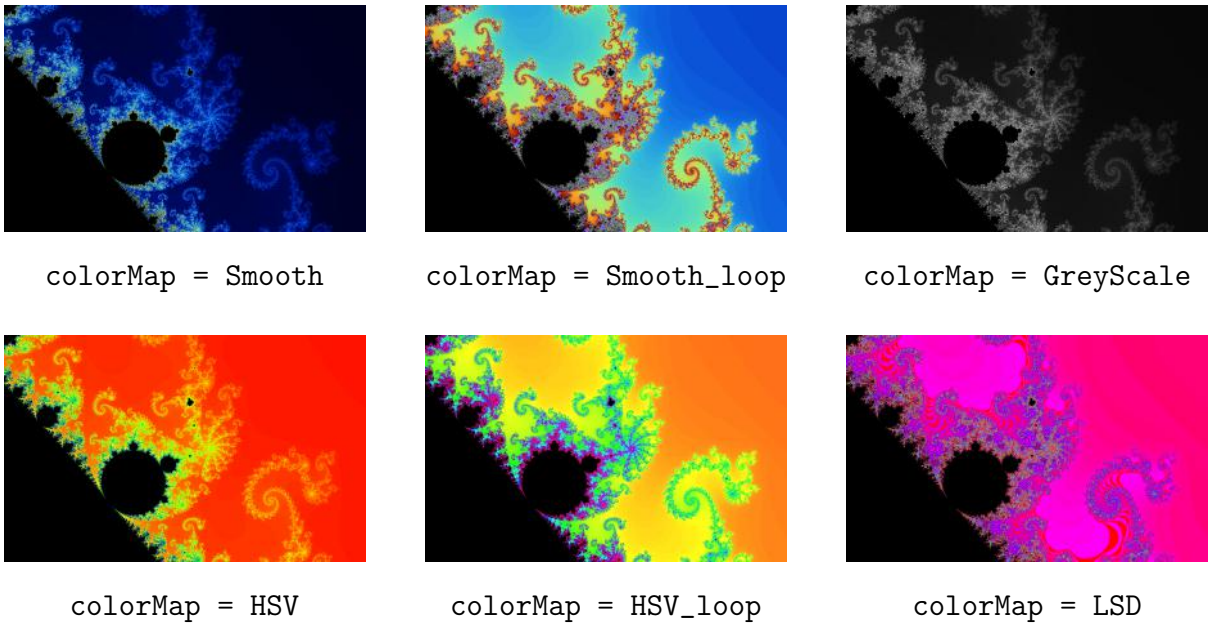


FIGURE 3.25 – Comparaison des principaux colorMaps développés

3.4.3 Fichiers de configuration

Afin d'apporter un maximum de flexibilité au programme, des fichiers de configurations ont été créés. Les principales options du programme telles que le type de convergence, le type de colorMap, les dimensions de la fenêtre, la position et le zoom de départ, et d'autres encore.

L'un des principaux intérés des fichiers de configuration est de pouvoir définir et redéfinir une configuration par défaut sans avoir à recompiler le programme.

La configuration par défaut a par exemple été décrite de la façon suivante :

Fichier de configuration pour la configuration par défaut

```
#Paramètres globaux
width = 1280
height = 800

#Affichage
colorMap = SMOOTH

#Convergence
convergence = DP_OMP_AVX+
```

Un autre intérêt de ces fichiers de configuration est de pouvoir définir des simulation, par un zoom continu sur un endroit spécifique de la fractale. On peut par exemple choisir le nombre d'itérations max, la position et le zoom de départ, le zoom de fin et le pas de zoom entre chaque image par exemple. On peut trouver ci-dessous un exemple de cas de simulation.

Exemple de fichier de configuration pour une simulation

```
#Paramètres globaux
width = 1280
height = 800

#Affichage
colorMap = smooth
maxIter = 512

#Convergence
convergence = DP_OMP_AVX+

#Auto zoom
autoZoom = on
nbSimu = 1

offsetX = -0.1598551935580027
offsetY = -0.6544325899631548

zoomStepTime = 0.0
zoomFactor = 1.05
startZoom = 0.004
finalZoom = 0.0000002952206096
```

Les fichiers de configuration ont un format inspiré du format ".ini". L'interprétation de ces fichiers par le programme a cependant été décrite par des fonctions faites maison, car il n'y a ici pas besoin de toutes les fonctionnalités apportées par les outils classiques de traitement de fichier ".ini".

On peut noter que tous les paramètres sont optionnels et que les erreurs dans les fichiers sont détectées et affichées dans la console à l'exécution. La casse n'a pas non plus d'importance.

3.4.4 Enregistrement de mesures automatique

Afin de comparer les performances des différentes implémentations logicielles, une fonctionnalité de mesure de temps d'exécution avec enregistrement automatique a été mise en place.

On peut réaliser sur un même scénario autant de mesure que l'on souhaite. A la fin des mesures, le temps d'exécution médian, moyen, min et max est calculé.

Il est possible d'exporter directement ces résultats dans un fichier ".csv". Il est possible de remplir le tableau en plusieurs fois (en plusieurs exécutions du programme), cela permet de pouvoir créer des "Shell Scripts" lançant exactement les mesures sur les scénarios désirées avec les paramètres désirés, et qui se feront donc à la suite.

Afin de garder une trace des conditions de mesure, le programme enregistre automatiquement dans le fichier ".csv" la référence du CPU et du système d'exploitation avec sa version.

3.4.5 Paramètres d'exécution

Le programme peut être lancé avec plusieurs paramètres de simulations. Si l'on désire utiliser un fichier de configuration, son nom doit être passé en premier argument. S'il n'est pas renseigné, la configuration du fichier `default_config` est utilisée.

L'utilisation des arguments d'execussion est détaillée dans la page de manuelle située dans le répertoire principal du programme, accessible via la commande `man ./fractal`, et donc un extrait peu être trouvé ci-dessous.

3.4.5.1 Paramètres d'exécution généraux

`-w` *width*
: integer x size of the window

`-h` *height*
: integer y size of the window

`-x` *offsetx*
: real x start position

`-y` *offsety*
: real y start position

`-i, --maxiter` *iter*
: maximum number of iteration, must be an integer

`-c, --close`
: close program after autozoom simulation

`-s, --save`
: If several simulations are done, `-f (--first)` and `-l (--last)` should be used so that the program can handle cumulative results.

`-d, --testID` *ID*
: If the saving option `-s (--save)` is used, a test ID should be added to tell the program which simulation is performed. It corresponds to the column in the .csv result output. The first column has an ID of 1. Must be an integer

`-f, --first`
: first element of a set of simulation. If used, the source file for result output is tmp.csv (cumulative result storing), otherwise results_template.csv is used.

`-l, --last`
: last element of a set of simulation. If used, the cumulative results are stored into a new file results_<date>_<time>.csv, instead of tmp.csv.

`--cute`
: enable a cute kitten mode

3.4.5.2 Paramètres de convergence

CPU - virgule flottante double précision

`--dp`
: double precision floatting point computation without any optimization.

`--dp_omp`
: double precision floatting point computation with OpenMP.

`--dp_omp_avx`
: double precision floatting point computation with OpenMP and AVX vectorization.

`--dp_omp_avx+`
: double precision floatting point computation with OpenMP, AVX vectorization and latency optimization.

`--dp_omp_avx++`
: double precision floatting point computation with OpenMP, AVX vectorization and experimental optimizations (actually slower than `dp_omp_avx`).

CPU - virgule flottante simple précision

`--sp`
: simple precision floatting point computation without any optimization.

`--sp_omp`
: simple precision floatting point computation with OpenMP.

`--sp_omp_avx`
: simple precision floatting point computation with OpenMP and AVX vectorization.

`--sp_omp_avx+`
: simple precision floatting point computation with OpenMP, AVX vectorization and latency optimization.

CPU - virgule fixe

`--fp`
: fixed point computation without any optimization.

`--fp_omp`
: sfixed point computation with OpenMP.

`--fp_omp_sse2`
: fixed point computation with OpenMP and SSE vectorization.

GPU - virgule flottante double précision

`--cuda_d`
: double precision floatting point computation using GPU.

`--cuda_d2`

double precision floatting point computation using GPU using double2.

`--cuda_d2_wp`
double precision floatting point computation using GPU using double2 without separated process function.

GPU - virgule flottante simple précision

`--cuda_s`
double precision floatting point computation using GPU.

3.4.5.3 Exemples de paramètres d'exécution

Commandes shell

```
# Commandes simples :
./bin/fractal --dp_omp_avx+
./bin/fractal -w 800 -h 600
./bin/fractal my_config.ini -w 800 -h 600
./bin/fractal -w 640 -h 480 -x -1 -y -0.5 -i 28
./bin/fractal --cuda -w 800 -h 600 -x -1 -y -0.5 -i 256
./bin/fractal auto_zoom1.ini --cuda_s
./bin/fractal auto_zoom1 --cuda_d2_wp --nbsimu 8
# Lancer une suite de simu :
./bin/fractal auto_zoom1 --nbsimu 2 --save --close --testID 1 --first
./bin/fractal auto_zoom2 --nbsimu 2 --save --close --testID 2
./bin/fractal auto_zoom3 --nbsimu 2 --save --close --testID 3 --last
```

4 Approche matérielle

Dans cette deuxième approche l'objectif est une fois encore d'implémenter la fractale de mandelbrot sur un écran. L'ensemble du travail réalisé a été pensé et implémenté sur une carte d'essai Nexys A7 développée par Digilent.
Le code source final se trouve sur le git dans le dossier *VHDL_mandelbrot_fonctionnel*.

4.1 Architecture

Dans un développement de circuit matériel une part non négligeable du temps de développement réside dans la création de l'architecture.

La première étape fut de penser un cahier des charges, pour ce faire plusieurs points sont ressortis :

— Affichage de la fractale sur un écran
— Possibilité de naviguer au sein de la fractale (déplacement et zooms)
— Calcul des itérations évolutif

Assez naturellement l'architecture suivante s'est dégagée :

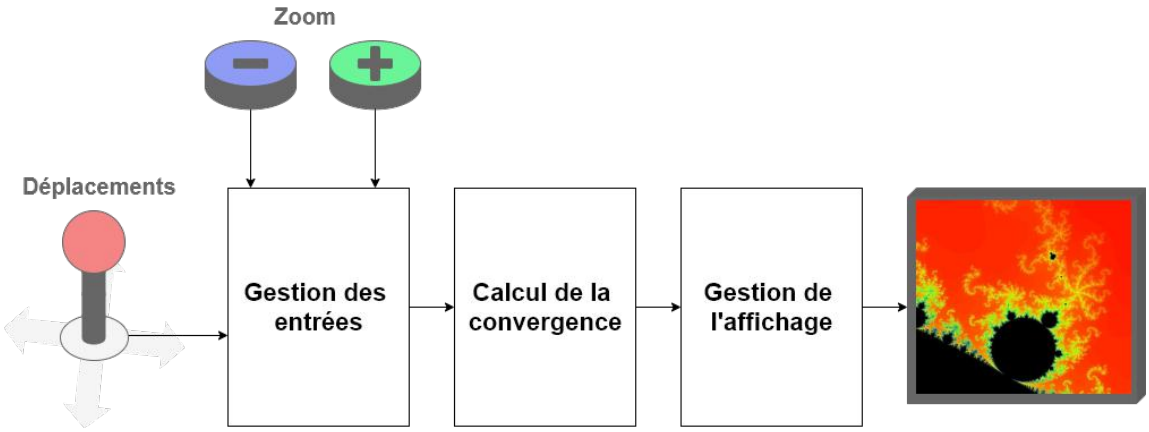


FIGURE 4.1 – Architecture simplifiée de la partie matérielle

L'architecture se décompose donc en trois grandes parties :

— **Gestion des entrées :** Cette partie gère les deux boutons poussoirs qui permettent de zoomer ou dézoomer et le déplacement sur l'axe x et y provenant du joystick [5].
— **Calcul de la convergence :** Cette partie est le coeur de l'implémentions, c'est ici que sont calculés l'ensemble des points de la fractale au travers d'un calcul de convergence.
— **Gestion de l'affichage :** Cette partie permet d'afficher les pixels précédemment calculés sur un écran doté d'un port VGA.

5. Modèle : Pmod JSTK : 2-axis Joystick

4.2 Gestion des entrées

Les entrées ajoutées au projets sont un simple plus, permettant à l'utilisateur de naviguer au sein de la fractale. Il aurait également pu être possible de crée des déplacements et zooms déterminés à l'avance ou encore d'afficher une seule image fixe. Cependant c'est pour nous, minimiser l'aspect visuel d'une telle fractale.

4.2.1 Le zoom

Pour le zoom deux boutons ont été utilisés. Chaque appui génère un front d'horloge modifiant la valeur du zoom précédent. Lorsque l'appui est maintenu des fronts d'horloges périodiques (50 ms) sont envoyés permettant un zoom continue plus agréable visuellement.

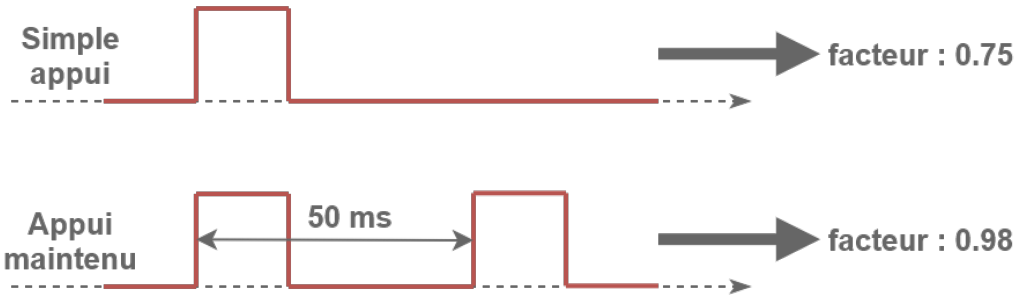


FIGURE 4.2 – Schéma de l'interaction utilisateur sur les boutons

Lorsque l'appui est maintenu le zoom revint à un facteur de 1.5/s et un dé-zoom de 0.7/s.

4.2.2 Le mouvement

Le déplacement se fait grâce au joystick. Il est connecté à un connecteur Pmod de la carte. Ce type de connectique supporte plusieurs protocoles tel que l'$I^2$2, l'UART et dans notre cas l'SPI [6] :

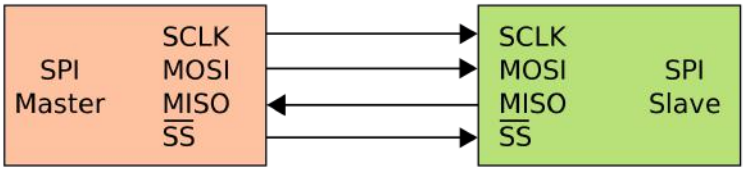


FIGURE 4.3 – Liaison SPI : un maître et un esclave

Digilent propose un module déjà fait pour utiliser le joystick. Après avoir supprimé ce qui n'était pas nécessaire, le module a été ajouté au projet. Les seuls données récupérées sont les valeurs (vecteurs de 10 bits) des deux axes. Ces données sont ensuite centré en zéro puis une zone "morte" autour du point central a été ajouté car le joystick n'est jamais exactement au centre. Cette zone couvre 20% de la plage totale des valeurs du joystick.

6. Serial Peripheral Interface

4.3 Calcul de la convergence

4.3.1 Synthèse de haut niveau HLS

La première partie du projet fut consacrée à l'utilisation de Vivado HLS afin de générer la partie liée à la convergence à partir du code natif C. Cependant lorsque la boucle *while* de l'algorithme fut rajoutée il semble que l'outil ne fonctionne plus et n'arrive pas à synthétiser les fichiers. Il a donc été décidé d'écrire directement les fichiers VHDL.

4.3.2 Synthèse de bas niveau VHDL

Pour calculer la convergence on a réalisé une architecture pipeline comportant un bloc générant les valeurs des points à afficher à la suite et une série de blocs permettants de réaliser les itérations.
On a donc un composant "point" qui prend en entrée le zoom et les offsets en X et Y et qui ressort la valeur de départ d'un point et son adresse sur l'écran VGA.

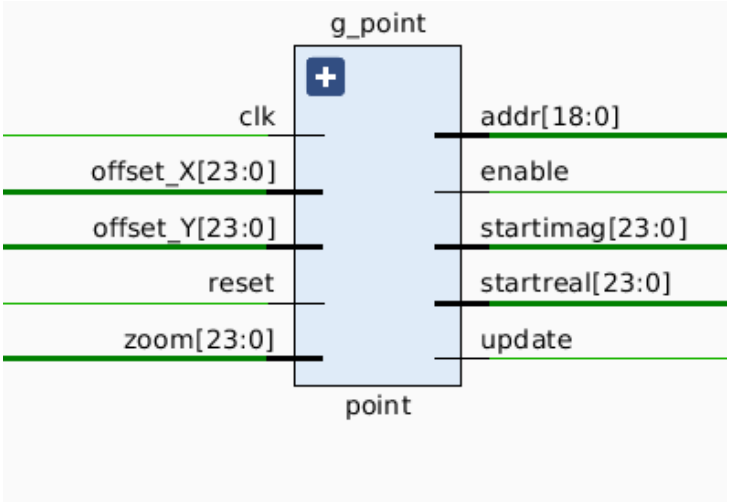


FIGURE 4.4 – Le bloc point

On a ensuite N composants "iteration" qui sont chacun à la suite l'un de l'autre et qui réalisent les calculs de la récurrence ainsi que la comparaison. Comme ils sont en pipeline ils doivent transmettre la valeur de départ, le nombre d'itération effectué jusqu'à présent ainsi qu'un signal "enable" servant à interdire le prochain composant à faire l'itération si la comparaison a été positive.

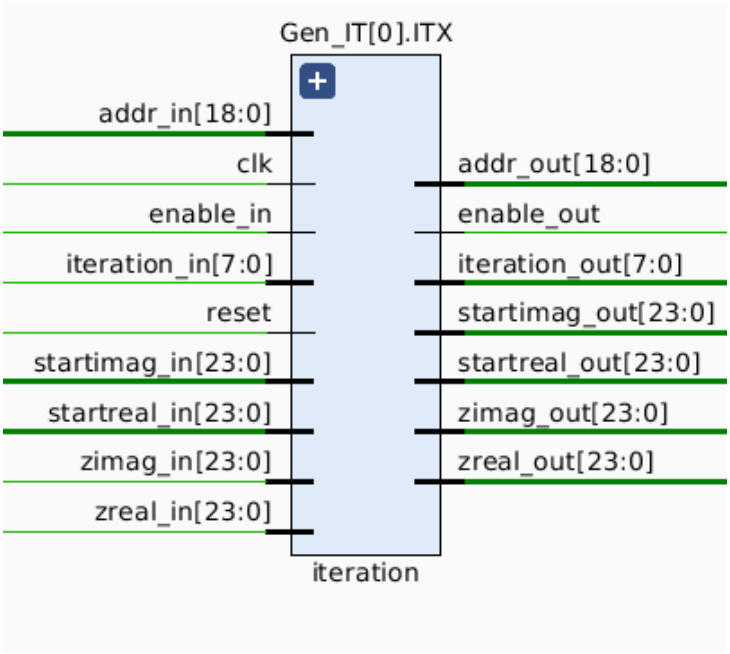


FIGURE 4.5 – Un bloc itération

Les variables sont initialement en virgule fixe sur 32 bits dont 25 pour la partie fractionnaire, pour avoir exactement la même chose que le code C sur la partie virgule fixe.

Un *generate* est utilisé pour décrire les blocs itérations. Il en est de même pour les signaux les entourant qui sont des array de std_logic ou de std_logic_vector de taille égale au nombre d'itérations.

4.4 Gestion de l'affichage

On utilise un colormap sur 8 bits chargé en RAM basé sur le module de M.Bornat. Pour chaque point, le nombre maximal d'itération correspond à l'adresse de la couleur stockée dans la RAM.
La sortie est sur écran VGA en 4/3 (640 pixels par 480).

4.5 Optimisation - Passage à des variables sur 24 bits

Avec des variables en virgules fixes sur 32 bits (dont 25 pour la partie fractionnaire) , avec 19 itérations on utilisait 93% des DSP disponibles dans la Nexys A7, à raison de 12 DSP par bloc "itération", ce qui nous empêche d'en rajouter.
Ce choix de taille des données avait été fait initialement pour avoir une correspondance avec le code C mais n'est pas forcement adapté pour FPGA car, d'une part les DSP n'ont que des entrées sur un maximum de 18 bits (ce qui signifie qu'on utilise plusieurs DSP pour un seul calcul), et, d'autre part, on n'a pas besoin d'une si grande précision pour l'affichage alors que ce qui nous restreint visuellement est le nombre d'itérations.
On a donc décidé de diminuer la taille des données à 24 bits (dont 18 pour la partie fractionnaire). On utilise alors, avec 19 itérations, seulement 50% des DSP tout en conservant le même aspect visuel, soit 6 DSP par bloc "itération".
On a donc pu à la suite augmenter le nombre d'itérations à 28 itérations.

4.6 Résultat

Le résultat suivant utilise 28 itérations avec des données sur 24 bits :

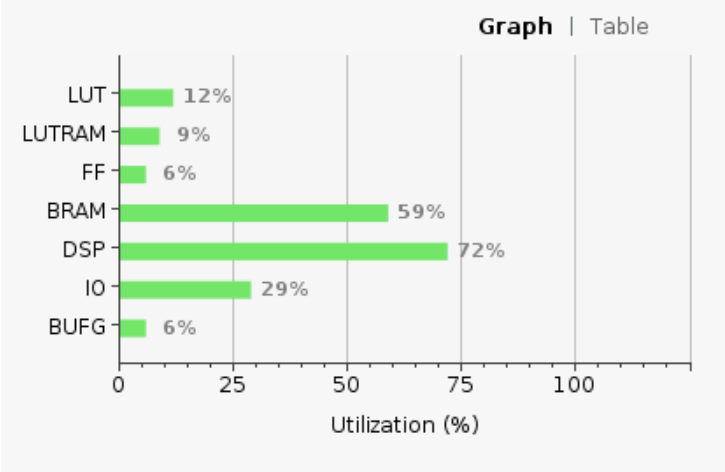
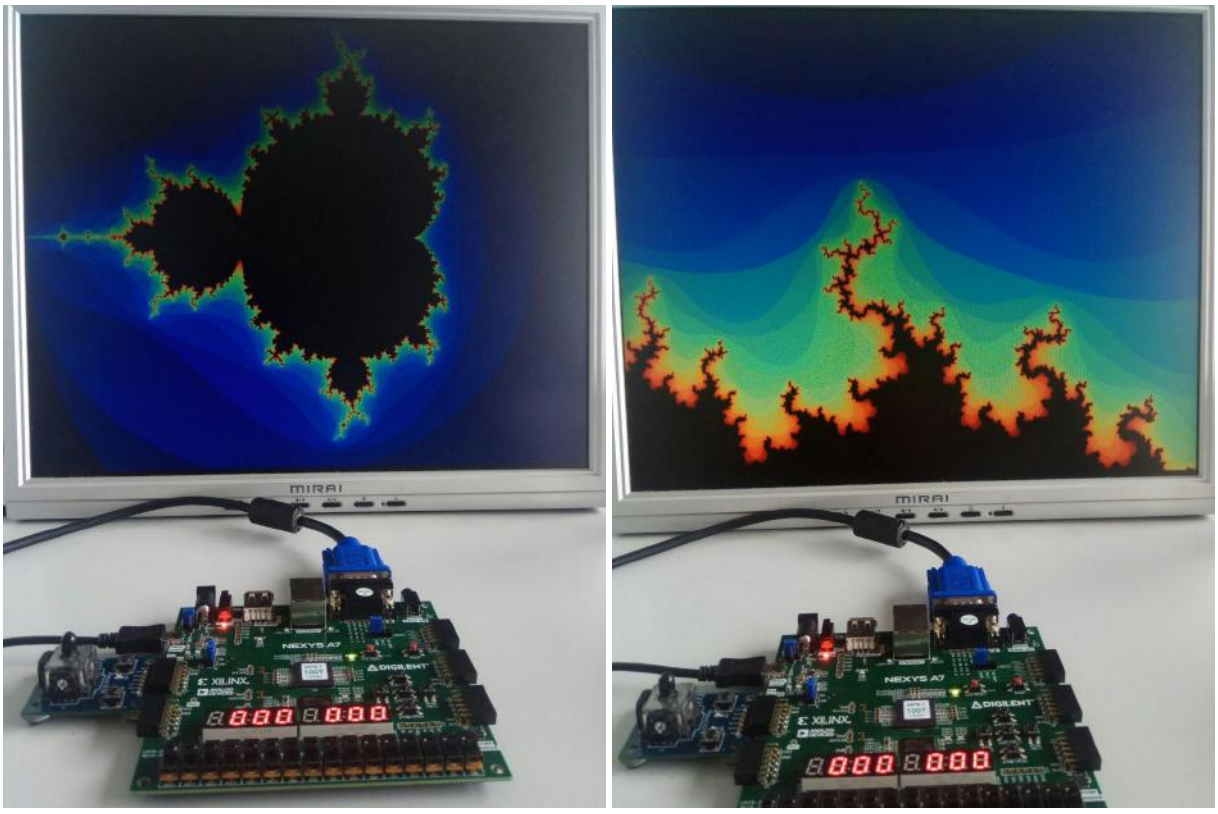


FIGURE 4.6 – Utilisation post-implémentation sur NexysA7

La puissance utilisée est de 0.404 W. On notera que l'on a cependant quelques problèmes de timing (WNS de -3.3 ns et TNS de -1487 ns pour une horloge à 100 MHz) sans que cela est toutefois un impact sur le résultat.

On a donc une fractale de Mandelbrot, se rafraîchissant à une fréquence de 100 MHz / (640*480), soit un temps d'affichage de 0.003072 s (par comparaison pour les mêmes caractéristiques [640*480 pixels et 28 itérations] on a entre 1 et 2 ms pour le CPU en utilisant des double et en étant entièrement optimisé).



5 Conclusion

Pour conclure finalement, on peut réaliser quelques comparaisons :

— La **vitesse d'exécution** du programme sur CPU (optimisé) est plus grande que celle utilisant le GPU pour la solution utilisant les Doubles, et cela malgré un programme bien parallélisable.
Cependant, le temps de développement d'une telle solution est relativement long comparé à celui du programme utilisant le GPU avec CUDA, et demande une connaissance poussée des architectures processeurs.

— Au niveau de la **vitesse d'exécution**, la solution du FPGA ne permet pas de concurrencer les 2 autres solutions : elle est très lente et possède un temps de développement très long. De plus elle demande le maniement de la virgule fixe.

— Pour les **performances énergétiques** : dans le cas d'utilisation des nombres flottants de 32 bits, le GPU se révèle très rapide puisque c'est sa spécialité. Mais il ne se révèle pas aussi rapide qu'espéré. Cependant, sa consommation (théorique) est tellement grande que même en utilisant les floats, cette solutions ne semble pas présenter d'avantage face au CPU sur le plan vitesse/puissance.
La solution sur FPGA permet cette fois d'avoir une consommation tellement réduite que son rapport vitesse/consomation est plus intéressant que celui des 2 autres solutions (en théorie puisque la puissance n'a jamais été véritablement mesurée).

On comprend qu'en ne prenant en compte que 3 critères qui sont la vitesse d'exécution, le temps de développement et la performance énergétique, la meilleure solution n'est jamais la même selon l'importance que l'on donne à chacun de ces 3 critères.

En réalité, dans l'industrie, il y a beaucoup plus de critères à prendre en compte, comme le coût (€) de la main d'oeuvre et du matériel, de la quantité d'erreur générées, etc. On comprend ainsi l'étendu des possibilités réalisables, et que même pour un cahier des charges bien fixé, plusieurs solutions peuvent être trouvées. De plus, il n'est pas forcement possible d'avoir une idée à l'avance de la solution répondant au mieux au cahier des charges.

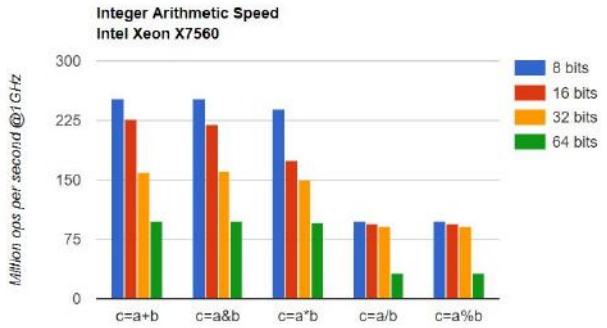
A Annexes

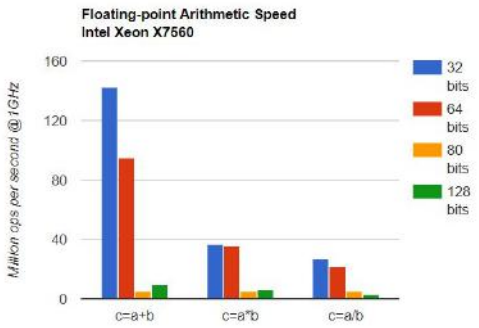
A.1 Approche logicielle

A.1.1 Performance des opérateurs arithmétiques

x86-64 Intel Nehalem

Intel Xeon X7560 - 2010 - 2.27GHz

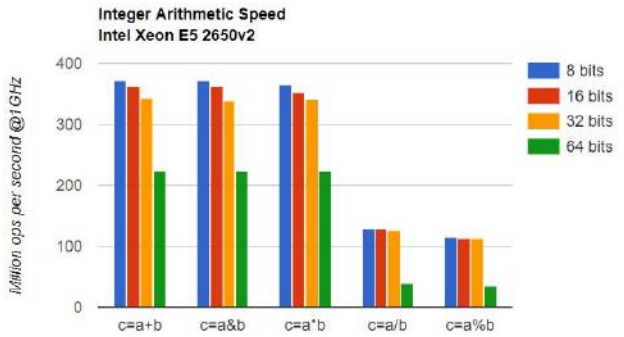


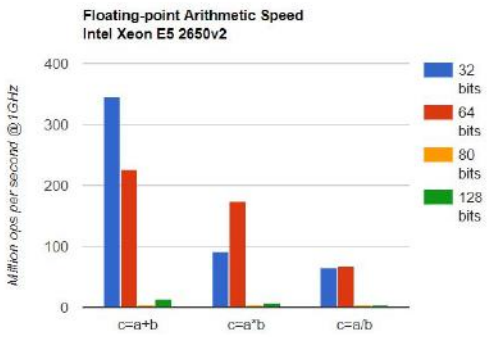


Compiler is `gcc 4.8` in Ubuntu 14.04.

x86-64 Intel Ivy Bridge

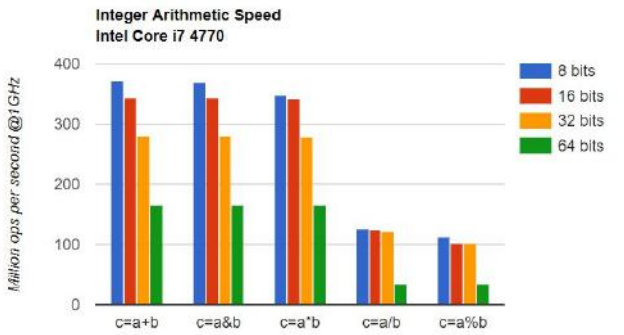
Intel Xeon E5 2650v2 - 2013 - 2.60GHz

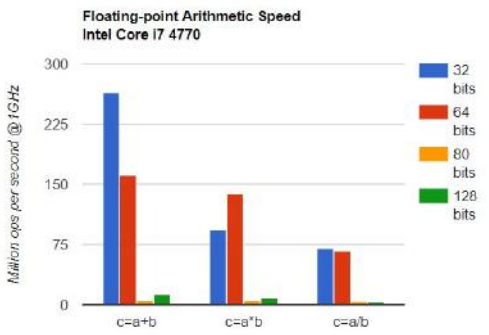




x86-64 Intel Haswell

Intel Core i7 4770 - 2013 - 3.40GHz





Compiler is `gcc 4.8` in Ubuntu 14.04.

source : http://nicolas.limare.net

A.2 Approche matérielle

A.2.1 Architecture RTL

Ci après, le schematic généré par Vivado :