X2012
06/03/2015

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Exploration de l'ensemble de Mandelbrot

INF560 : Calcul parallèle

Etienne Ferrier, Gurvan L'Hostis

etienne.ferrier@polytechnique.edu, gurvan.lhostis@polytechnique.edu

Abstract: Ce rapport décrit la méthode que nous avons développée pour explorer l'ensemble de Mandelbrot grâce à une implémentation GPU.
Nous montrons dans un premier temps comment paralléliser le calcul des pixels, puis nous décrivons la librairie de flottants de précision arbitraire parallélisée que nous avons implémentée pour le calcul fractal.

Keywords: Fractal exploration • GPU • CUDA • XaoS • Arbitrary-precision arithmetic

1. Introduction

L'objectif de ce projet est d'implémenter en CUDA un outil capable de zoomer de façon fluide dans l'ensemble de Mandelbrot[1], à l'instar du freeware Xaos[2][3] (CPU). Le calcul GPU permet à la fois de distribuer les calculs sur tous les pixels mais aussi d'effectuer plus rapidement les opérations sur des flottants de précision arbitraire et donc d'explorer plus loin la fractale.

2. Enjeux du calcul de l'ensemble de Mandelbrot

Le célèbre ensemble de Mandelbrot est calculé de la façon suivante : on se place dans le plan complexe, et à chaque point c on associe la suite

$$\begin{cases} z_0 = 0 \\ z_{i+1} = z_i^2 + c \end{cases} \tag{1}$$

L'ensemble de Mandelbrot est l'ensemble des points dont la suite associée ne diverge pas.

[1] *http://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot*
[2] *https://www.youtube.com/watch?v=9ltIvooYa1U*
[3] *http://matek.hu/xaos/doku.php*

En pratique, on réalise des visualisations à partir de cette définition en faisant correspondre des coordonnées complexes aux pixels d'une image et en appliquant cette suite. Les points pour lesquels une itération de la suite dépasse un module de 2 vont nécessairement diverger, c'est pourquoi l'on utilise cette barrière comme critère d'affichage. On se fixe une limite d'itérations à effectuer au maximum, on considère que les pixels qui atteignent cette limite sont dans l'ensemble et on les colore en noir tandis qu'on attribue aux autres une couleur correspondant à leur *temps d'échappement*.
Une première source de parallélisation apparait ici comme évidente : les suites correspondant aux pixels sont indépendantes et l'on peut donc appliquer une méthode de type *embarrassingly parallel*.

Le calcul de l'ensemble de Mandelbrot à petite échelle, c'est-à-dire des zooms profonds, présente lui deux difficultés. La première difficulté provient de la nature même de notre méthode de visualisation. On doit définir une borne au nombre d'itérations que l'on explore, mais puisqu'on cherche généralement à zoomer sur la bordure de l'ensemble il existe toujours une certaine profondeur à partir de laquelle cette borne est insuffisante. En effet, la plupart des points qui sont hors de l'ensemble (et qui donnent la couleur à l'image) seront à tort considérés comme appartenant à l'ensemble. La bordure perd son relief. Ainsi, plus on veut aller loin, plus il faut faire d'itérations.
La seconde difficulté provient de la précision de calcul. En effet, l'approximation en nombres flottants des coordonnées que l'on fait en passant du point abstrait au pixel n'est suffisante que jusqu'à une certaine échelle. À terme, tous les points ont les mêmes coordonnées flottantes et donc le même temps d'échappement. On n'observe alors plus de détail. Il n'existe qu'un seul moyen de contrer ceci : augmenter la précision des calculs.
C'est là qu'intervient l'aspect de parallélisme intéressant du projet. Nous avons développé notre propre librairie d'arithmétique parallélisée de grands flottants dédiée au calcul de fractales, ce qui nous permet de faire certaines hypothèses intéressantes dans son implémentation.

3. Calcul des pixels en parallèle

Notre implémentation du calcul de fractales contient quatre méthodes, de la plus simple à la plus évoluée : calcul sur flottant en CPU, calcul sur flottant en GPU, calcul avec grands flottants sur CPU, calcul avec grands flottants parallélisés sur GPU.

Calcul sur CPU

Le calcul d'images fractales en CPU consiste à traduire directement l'idée décrite dans la partie précédente en code. Nous avons implémenté une méthode de zoom interactive permettant de cliquer sur un point de l'image pour zoomer vers celui-ci. Les seuls paramètres que l'on se donne sont la résolution de l'image, le facteur de zoom par clic et le nombre maximum d'itérations que l'on calcule pour chaque pixel.

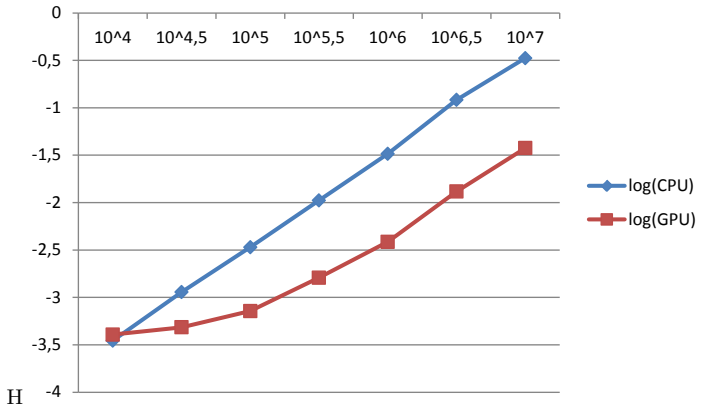


Figure 1. *Logarithme du temps de calcul en secondes en fonction du nombre de pixels à calculer sur le CPU et sur le GPU en précision simple. On observe qu'une certaine masse de calculs à effectuer est nécessaire pour rendre intéressant d'utilisation du GPU.*

Calcul sur GPU

L'algorithme de calcul d'image fractale est de type *embarrassingly parallel*, c'est-à-dire que la parallélisation est évidente et immédiate.

Il suffit en effet d'écrire une fonction `_device_` pour implémenter le processus itératif à partir d'un point et une fonction `_global_` qui appelle cette fonction sur chacun des couples de coordonnées des pixels de l'image que l'on souhaite obtenir.

Nous utilisons pour cette méthode des blocs de taille 32x32 mais nous n'utilisons pas la mémoire partagée. Le calcul des itérations successives ne nécessite que cinq variables flottantes et un compteur entier, soit au total $32*32*6*4 = 24576$ octets, ce qui est compatible avec la plupart des cartes graphiques (celles que nous utilisons possèdent 65536 octets de mémoire registre par bloc).

Comparaison des performances

Les temps de calculs des deux implémentations en fonction du nombre de pixels sont représentés en figure 1. On observe qu'à partir d'un certain seuil, le rapport du temps de calcul du CPU sur le GPU se stabilise à 10.

Le GPU en simple précision permet d'obtenir un zoom en temps réel sur des images de très haute définition.

4. Arithmétique de précision arbitraire

La précision flottante sur 32 bits n'est pas satisfaisante pour l'exploration de fractales. Une de leur propriétés principales est l'autosimilarité de l'ensemble quand on descend dans les échelles. S'arrêter à 10^{-10} reste très limité.

C'est pourquoi nous avons voulu, comme dans la plupart des projets d'exploration de fractale, se doter de nombres de précision arbitraire. Nous détaillons dans cette partie les algorithmes que nous avons utilisés pour aboutir à une implémentation d'opérations arithmétiques de précision arbitraire spécifiques au calcul d'images fractales.

bool	uint32_t	uint32_t	uint32_t
±	0	1	2
	2^0	2^{-32}	2^{-64}
+	3	0x80000000	0

Figure 2. *Structure d'un grand flottant de taille 3. Premiere ligne : type des constituants du grand flottant. Deuxieme ligne : représentation utilisée dans ce rapport. Troisième ligne : poids des décimales. Quatrième ligne : exemple de l'encodage du nombre +3,5.*

Les hypothèses sur lesquelles on peut s'appuyer dans la conception d'une librairie de grands flottants pour le calcul d'image fractales sont les suivantes :

- Les flottants sont bornés par 2^{32}. En effet, puisque l'on arrête d'itérer la suite dès lors que l'on dépasse un module de 2, il ne sera jamais nécessaire de considérer des nombres plus grands que quelques unités.
- Les nombres considérés sont de l'ordre de grandeur 10^0. Il sera très rare de rencontrer des nombres dont les premières décimales sont nulles.

Ces deux hypothèses permettent de ne pas considérer d'exposant pour nos flottants, on est dans une situation similaire à celle d'une librairie pour nombres entiers. La représentation mémoire que nous utilisons est illustrée dans la figure 2. Un nombre est représenté par un booléen pour le signe et un tableau de m entiers non signés de 32 bits -que nous appelons décimales par abus de language- pour les chiffres de l'écriture en base 2^{32}.
Il est à noter que nos nombres ne sont pas des flottants à proprement parler ; on garde cependant cette appelation de grand flottant pour les distinguer des nombres entiers.

L'addition se fait de façon triviale en additionnant les chiffres un-à-un et en propageant la retenue. La multiplication est résumée par ces quelques lignes :

$$x = \sum_{i=0}^{m-1} x_i b^{-i} \text{ avec } \forall i, 0 \leq x_i < b \tag{2}$$

$$y = \sum_{j=0}^{m-1} y_j b^{-j} \text{ avec } \forall j, 0 \leq y_j < b \tag{3}$$

$$xy = \sum_{i=0}^{m-1} x_i b^{-i} y \tag{4}$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_i b^{-i} y_j b^{-j} \tag{5}$$

$$= \sum_{k=0}^{m-1} \left(\sum_{i+j=k} x_i y_j \right) b^{-k} \tag{6}$$

$$(xy)_k = \sum_{i+j=k} x_i y_j \mod b + \left(\left(\sum_{i+j=k+1} x_i y_j \right) /b \right) \mod b + \left(\sum_{i+j=k+2} x_i y_j \right) /b^2 \tag{7}$$

Il est à noter que le passage de la ligne (5) à la ligne (6) se fait en arrondissant à la précision utilisée.
Les opérations modulo b s'appuient sur la gestion de *integer overflow*, et il y a quelques astuces de calcul détaillées dans le code pour gérer les multiplications chiffre par chiffre ainsi que la propagation des retenues.

5. Opérations de précision arbitraire en parallèle

Afin d'accélérer les calculs, nous avons parallélisé les opérations arithmétiques (addition et multiplication) sur les grands flottants. Nous avons conservé la structure de données `bool + uint32_t[m]` (où m est le nombre de décimales) utilisée en CPU mais en utilisant autant de threads que de décimales afin de distribuer les opérations. Nous avons donc adapté les méthodes décrites dans partie précédente au calcul GPU, ce qui nous a permis de passer d'une complexité de $O(m)$ à $O(1)$ pour l'addition et de $O(m^2)$ à $O(m)$ pour la multiplication.

5.1. Addition distribuée

L'addition est effectuée en deux étapes (voir Fig 3). Dans la première (*Addition step*), chaque thread d'indice k réalise l'addition des décimales k de A et de B, écrit le résultat dans C et calcule s'il y a une retenue. Dans la seconde (*Carry step*), chaque thread ajoute l'éventuelle retenue à la décimale supérieure de C. À la différence de l'addition en CPU, la retenue n'est pas propagée plus d'une fois. Dans le cas d'une soustraction, celle-ci est d'abord réalisée en supposant |A| > |B|. Les threads d'indice 0 et 1 vérifient ensuite s'il y a eu un changement de signe lors de la soustraction. Si c'est le cas, la soustraction est recalculée avec |B| > |A|.

Cette méthode d'addition produit deux types d'imprécisions. Le premier provient du fait que la retenue n'est propagée qu'à la décimale supérieure soit une propagation sur un nombre de bits entre 32 et 64 dans l'écriture

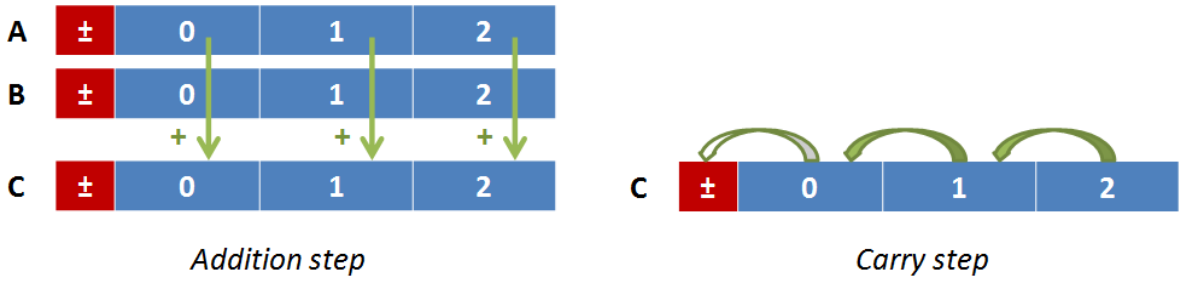


Figure 3. *Addition distribuée pour $m = 3$. $C = A+B$. L'action de chaque thread est représentée par une flèche verte.*

binaire du grand flottant. La deuxième imprécision provient du fait que le signe d'une soustraction est déterminé seulement par la différence entre les deux premières décimales de chaque grand flottant. Cela signifie que la soustraction de deux grands flottants proches à 2^{-32} près sera faussée. Cependant, ces deux imprécisions sont nécessaires afin de distribuer efficacement les calculs et ne produisent qu'une très faible probabilité d'erreur lors du calcul de l'ensemble de Mandelbrot, étant donné la distribution de grands flottants utilisée.

5.2. Multiplication distribuée

Comme l'addition, la multiplication est effectuée en deux étapes (voir Fig 4). Dans la première (*Partial multiplication step*), chaque thread d'indice k réalise la somme partielle des produits des décimales i de A et j de B telles que $i + j = k$. Ce résultat est une somme partielle de nombres de 64 bits, il est donc décomposé en trois parties : `little`, `big` et `carry`. `Little` correspond aux 32 premiers bits, `big` aux 32 suivants, et `carry` à la retenue totale sur 8 bits provenant de la somme partielle. Dans la seconde étape (*Sum step*), ces sommes partielles sont ajoutées au résultat en trois temps : `little`, puis `big`, puis `carry`.

La retenue n'est propagée que de `big` à `carry` mais puisque `carry` $\leq m+1 \ll 2^{32}$, l'erreur introduite est négligeable sur la distribution de grands flottants considérée. C'est aussi puisque $m + 1 < 2^8$ que `carry` peut être encodé sur seulement 8 bits. Etant donnée notre méthode de multiplication, il est nécessaire d'avoir A, B et C en mémoire partagée dans notre implémentation. Cependant, `little`, `big` et `carry` peuvent être stockés en registre.

5.3. Détails d'implémentation

Dans notre implémentation, ces opérations sont effectuées en place (fonctions device `addIP` et `multIP`), ce qui permet de limiter l'allocation de mémoire partagée. Les algorithmes en place sont sensiblement les mêmes que ceux décrits précédemment.

Nous avons également limité le nombre d'accès à la mémoire partagée en utilisant au maximum les registres. Le code de ces fonctions est visible dans le fichier `BigMandelGPU.cu`. Ces opérations sont notamment utilisées

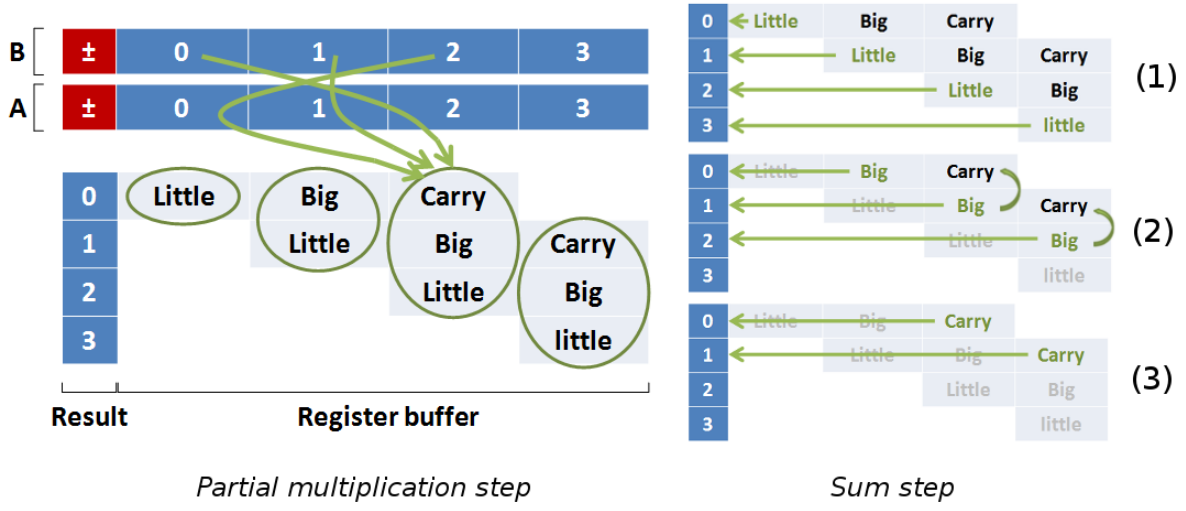


Figure 4. *Multiplication distribuée pour $m = 3$. $C = A \times B$. À gauche, chaque thread calcule le contenu d'un cercle vert. La multiplication partielle réalisée par le thread d'indice 2 est représentée par les flèches vertes. A droite, chaque thread participe à la somme finale en 3 étapes. L'action de chaque thread est représentée par une flèche verte.*

dans les fonctions device `complexSquare`, `testSquare` et `loadStart` qui permettent d'implémenter respectivement l'opération $Z \leftarrow Z^2$, le test $|Z|^2 < 4$ et le chargement du point de départ de l'itération principale. Nous avons optimisé ces fonctions en limitant à la fois le nombre de valeurs intermédiaires en mémoire partagée et le nombre de multiplications utilisées (voir le code de `complexSquare`).

6. Calcul de l'ensemble de Mandelbrot en parallèle

Afin de calculer l'ensemble de Mandelbrot en parallèle, nous avons mis en place une structure adaptée de threads et de mémoire partagée sur laquelle nous effectuons le calcul de chaque pixel de l'image.

6.1. Description d'un bloc

Nous avons choisi d'utiliser des blocs de taille (32, 1, m). Chaque bloc est donc constitué de 32 colonnes de taille (1, 1, m) (voir Fig 5). Chacune de ces colonnes est responsable du calcul de la couleur d'un pixel et contient m threads qui lui permettent d'effectuer les opérations arithmétiques décrites dans la partie précédente. Le thread de coordonnée $z = 0$ de chaque colonne est appelé le thread de tête et est utilisé pour effectuer les opérations qui ne peuvent pas être distribuées.

6.2. Description de la grille

Comme dans la première implémentation GPU, les dimensions en threads de la grille en x et y correspondent aux dimensions de l'image calculée (voir Fig 6). Cependant, nous utilisons ici une colonne et non un unique thread

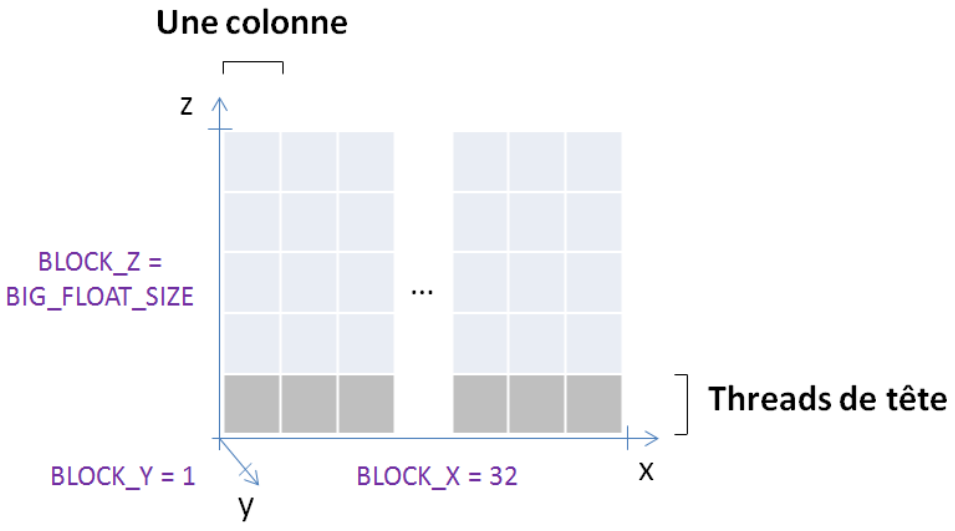


Figure 5. *Représentation d'un bloc. Une colonne est un ensemble de m = BIG_FLOAT_SIZE threads de même coordonnées x et y. Le thread z = 0 de chaque colonne est appelé thread de tête.*

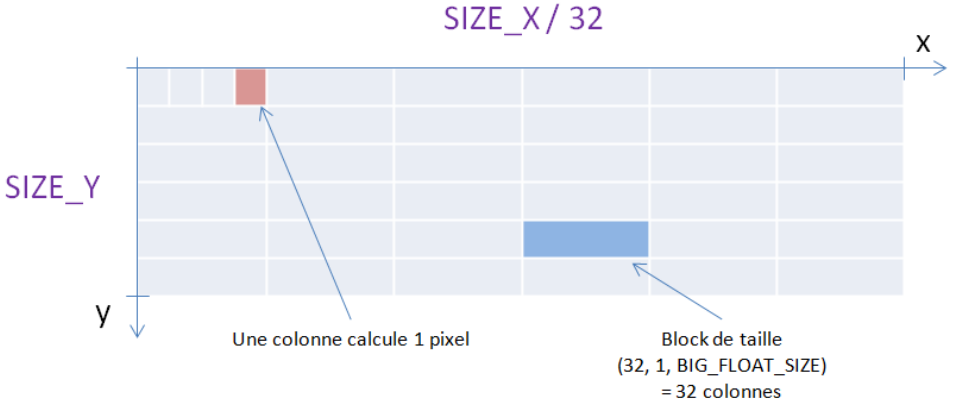


Figure 6. ***Représentation de la grille. SIZE_X (multiple de 32) et SIZE_Y sont les tailles de l'image calculée.***

afin de calculer la couleur de chaque pixel. Nous avons essayé plusieurs dimensions de blocs et de grilles avant de choisir le pavage par blocs de taille (32, 1, m).

6.3. Mémoire partagée

Afin d'implémenter la boucle principale du calcul de l'ensemble de Mandelbrot, nous avons alloué 5 grands flottants à chaque colonne (voir Fig 7). Les grands flottants Xinit et Yinit contiennent la valeur constante qui reflète la position du pixel dans le plan complexe. X et Y contiennent les valeurs successives de la suite complexe principale. Enfin, Tmp est utilisé afin de réaliser l'opération $Z \leftarrow Z^2$.

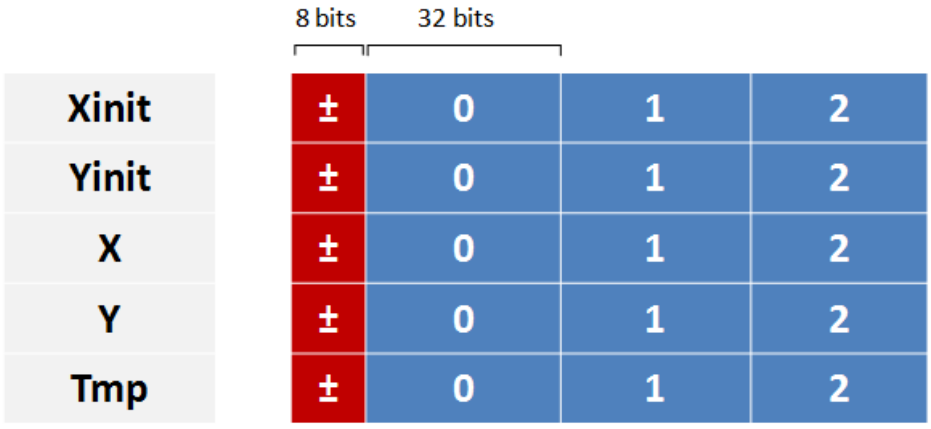


Figure 7. *Grands flottants alloués en mémoire partagée pour chaque colonne.*

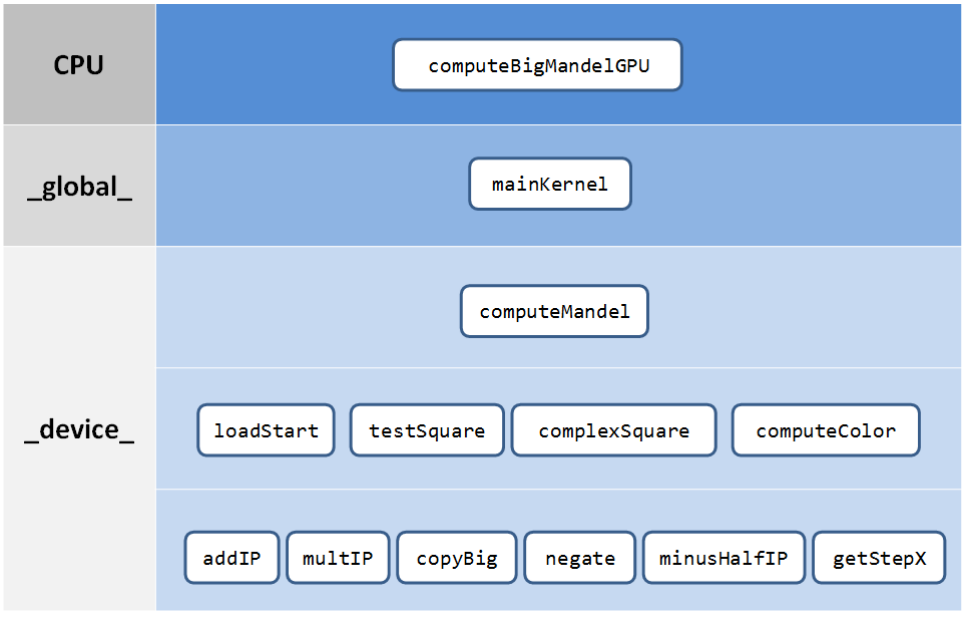
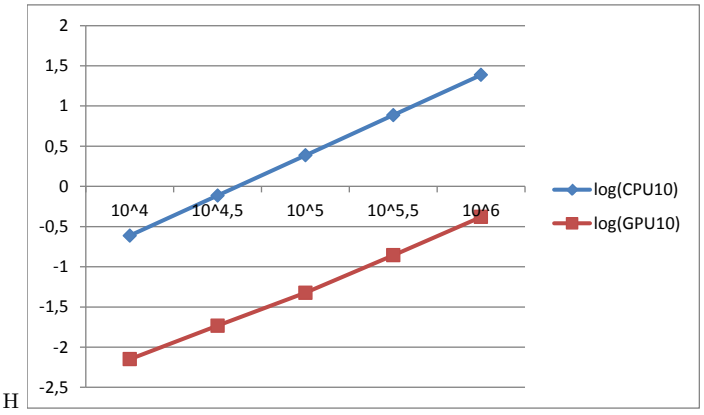


Figure 8. *Structure hiérarchique de notre pipeline. La première colonne donne la portée des fonctions. La deuxième colonne situe chaque fonction GPU dans la hiérarchie. Ces fonctions sont codées dans le fichier* `BigMandelGPU.cu`.

Au total, ce sont donc $160 + 640 * m$ octets qui sont alloués en mémoire partagée pour chaque bloc. On peut remarquer que comme le calcul de chaque pixel est indépendant, ces 5 grands flottants n'ont en réalité pas besoin d'être partagés à tout le bloc mais seulement à leur colonne.



H

Figure 9. *Logarithme du temps de calcul en secondes en fonction du nombre de pixels à calculer sur le CPU et sur le GPU à 10 entiers de précision (soit 40 octets).*

6.4. Structure du pipeline GPU

Notre pipeline est organisé en 5 niveaux (voir Fig 8). Au niveau CPU, les paramètres de la partie du plan complexe à calculer sont envoyés au GPU. Au niveau `_global_`, toute la mémoire partagée est allouée et initialisée puis les threads sont différenciés selon le pixel qu'ils représentent (i.e. leur position en x et y). Le premier niveau `_device_` implémente la boucle principale puis calcule la couleur de chaque pixel en fonction du résultat des itérations. Le deuxième niveau `_device_` implémente la fonction de coloration ainsi que les opérations qui demandent l'utilisation de manière astucieuse de plusieurs opérations arithmétiques simples. Enfin, le troisième niveau `_device_` implémente les opérations arithmétiques simples. La majeure partie de la synchronisation est codée à ce niveau.

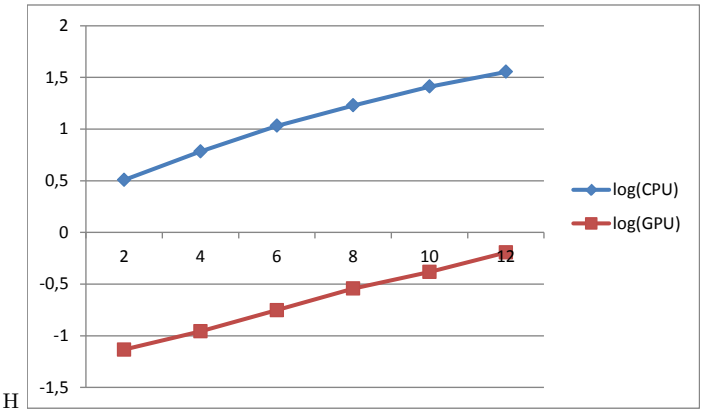
6.5. Performances

Une comparaison des performances CPU/GPU sur diverses résolutions est disponible en figure 9. Cette fois-ci le GPU est efficace dès les faibles résolutions. On compare également les performances en fonction du nombre d'entiers utilisés pour représenter les flottants en figure 10. On observe un rapport de temps de calcul d'environ 60. Cette augmentation du rapport montre que la parallélisation de l'arithmétique de précision arbitraire que nous avons implémentée apporte un gain de vitesse important.

7. Développements possibles

Algorithme de Karatsuba

On pourrait améliorer notre vitesse de calcul en utilisant l'algorithme de Karatsuba qui permet la multiplication en $O(n^{log_2(3)})$ grâce à l'astuce de calcul suivante :



H

Figure 10. *Logarithme du temps de calcul en secondes en fonction du nombre d'entiers de précision sur le CPU et sur le GPU pour* 10^6 *pixels.*

$$(a * 10^k + b)(c * 10^k + d) = ac * 10^{2k} + (ac + bd - (a - b)(c - d))10^k + bd \tag{8}$$

Pour le calcul en parallèle, cette astuce de calcul permet d'utiliser moins de threads pour une multiplication tout en conservant un temps $O(m)$, ce qui améliore donc les performances globales de l'implémentation.

Algorithme de XaoS

Notre programme répond à la même problématique que le freeware XaoS. Ce dernier, bien qu'étant programmé en CPU, permet d'obtenir un zoom fluide dans l'ensemble de Mandelbrot[4]. En fait, l'algorithme de XaoS détermine de façon astucieuse un grand nombre de lignes et de colonnes de l'image qui peuvent être conservées et recopiées lors du zoom, ce qui permet de réduire significativement le nombre de pixels à calculer. Il serait possible d'effectuer la même astuce dans notre programme afin de réduire davantage le temps de calcul lors du zoom.

[4] *XaoS est basé sur une double précision et non une précision arbitraire.*