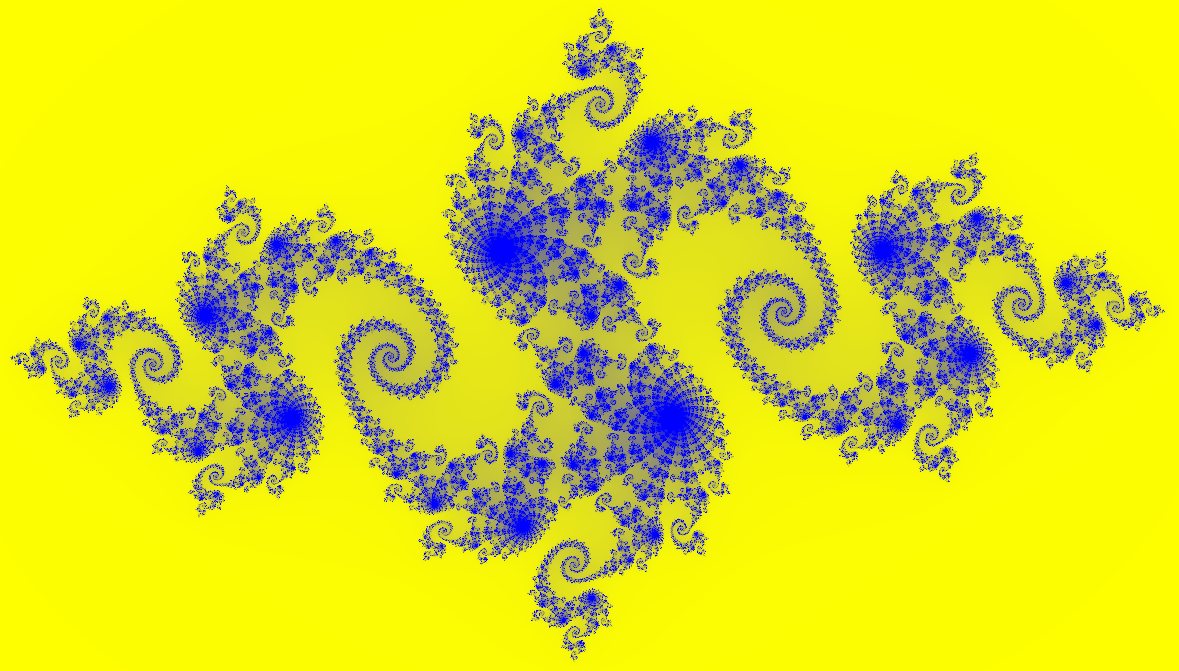
**Animation sur les ensembles de Julia grâce à CUDA**



# Projet

## Principe des ensembles de Julia et algorithme

A un complexe z=x+i.y, on associe une couleur (ou une intensité de couleur) selon l’algorithme suivant :

COULEUR <-- 0  
Z<--x+i.y  
Répéter  
        Z <-- Z² + C  
        COULEUR <-- COULEUR+1  
Jusqu'à ce que Module(Z)>2

Cela permet alors de créer des images fractales, lorsqu’on associe à chaque pixel des coordonnées (x,y). Cet algorithme est alors très facilement parallelisable car à chaque coordonnée, on doit faire les calculs de l’algorithme précédent indépendamment des autres pixels. Mon projet a alors été d’utiliser un GPU NVidia via CUDA pour réaliser ces calculs en espérant que ce dernier soit assez performant pour alors réaliser une animation en interaction avec l’utilisateur. De plus, cela a été réalisé dans une optique d’exploration de CUDA afin de trouver la solution optimale.

## Développement du projet

Dans un premier temps, j’ai réalisé une animation simple avec une interaction limitée avec l’utilisateur afin de comparer les performances selon la méthode utilisée. Cette animation avait une résolution de 256x256 pixels, permettait de se déplacer dans l’image (droite/gauche/haut/bas et de zoomer/dézoomer) et les calculs étaient réalisés avec une précision float.

1. D’abord, j’ai mis au point cette animation de manière classique sans l’utilisation de CUDA, avec tous les calculs réalisés par le CPU via une boucle sur les pixels. Cela m’a permis d’avoir le squelette de mon programme. On obtenait alors un frame rate entre 4 et 5 (en haut à gauche de l’image) :

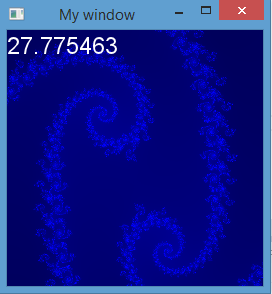


1. En utilisant le GPU :

L’utilisation de CUDA pour traiter des tâches parallèles se fait via une fonction kernel. On lui renseigne le nombre de threads par block et le nombre de blocs par grille. Alors la fonction appliquera le même code à tous les threads qui seront alors identifiés par son numéro dans le bloc et par le numéro de son block. Un premier essai a été fait en indiquant un thread par bloc (soit autant de bloc que de pixel) avec un résultat dans les mêmes conditions que précédemment autour de 0.8 frame par seconde :



Après un peu de déception, j’ai alors essayé d’attribuer 16\*16 threads par bloc et ce qu’il fallait alors de blocs par grille, ce qui a donné un résultat bien plus probant avec un frame rate de 28 :



Ensuite j’ai essayé de mettre les paramètres de la génération des images (le complexe C de l’algorithme, la position de l’image, et l’échelle (associées au déplacement dans l’image), dans la mémoire constante du GPU. Cette mémoire permet un accès plus rapide aux données en échange de les déclarer constantes, c’est-à-dire qu’elles ne pourront être modifiées par le GPU. En effet, la performance des calculs dépend généralement plus des temps d’écriture et de lecture plutôt que des performances des processeurs de la carte graphique.

L’utilisation de cette mémoire a nécessité son lot de changements par rapport au programme classique, notamment à cause du fait qu’on ne peut pas déclarer une variable constante si elle appartient à une classe qui admet un constructeur ou un destructeur. Ainsi j’ai dû me passer de la classe complexe que j’avais créée. Heureusement, ce ne fut pas en vain :



Enfin, j’ai essayé de mettre ces paramètres non plus en mémoire constante mais en liant à une texture, qui est un autre type d’accès mémoire qui peut de même améliorer les performances. Mais ce ne fut pas le cas ici :



Ainsi, j’ai gardé l’utilisation de la mémoire constante pour développer mon projet. De plus, j’ai eu la bonne surprise de me rendre compte que le frame rate de mon expérience était limité par l’obligation d’appuyer sur une touche pour déclencher le calcul de l’image. Ainsi quand je suis passé en calcul permanent de l’image, le frame rate est alors passé à plus de 150 et pour des résolutions plus grandes. En effet, la performance du frame rate dépend bien sûr de la résolution mais surtout du nombre de points où la couleur de dessin est intense dans l’image.

**Remarque :** Après des heures de recherches, en passant initialement par une phase de doute et de questionnement sur la possibilité de mener à bien ce projet vu l’impossibilité de compiler la moindre ligne de code, je n’ai pas réussi à découper le programme en plusieurs fichiers de manière à ce que le compilateur de CUDA fonctionne. Ainsi tout le code est contenu dans un fichier (juliaFloat.cu ou juliaDouble.cu, qui sont identiques à la déclaration du type des variables près).

# Projet final et guide d’utilisation

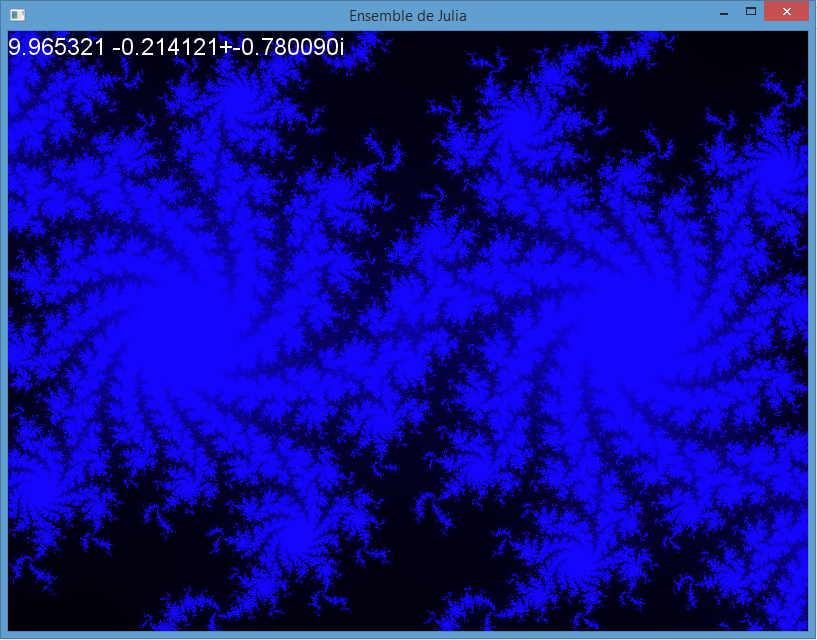
## Description du projet final

Le rendu final est une animation en interaction avec l’utilisation d’ensembles de Julia. Comme précédemment l’utilisateur peut se déplacer dans l’image. De plus il peut modifier la couleur de dessin utilisé dans l’algorithme, ainsi que la couleur du fond. De plus l’utilisateur peut initialement choisir la résolution de l’image. Enfin ce rendu existe sous deux versions : l’un avec une précision float dans le fichier juliaFloat.exe qui admet alors un zoom plus limité mais de meilleurs performances que l’autre version, la version avec une précision double contenu dans le fichier juliaDouble.exe.

## Guide d’utilisation

A l’ouverture du fichier, va vous être demandé la résolution que vous voulez afficher. Rentrer R pour définir ensuite une largeur et une hauteur de résolution. Sinon rentre n’importe quoi d’autre pour être en plein écran.

En haut à gauche sont indiqués le frame rate et la valeur actuelle de la constante complexe.



### Touches :

#### Déplacements dans l’image :

Touches flèches pour se déplacer vers le haut/bas/droite/gauche  
Shift Gauche ou NumPad8 : Zoom +  
Ctrl gauche ou NumPad2 : Zoom -

#### Modification de la constante complexe :

Z/Q/S/D : Déplacement de la constante dans le plan complexe

#### Changement des couleurs :

Couleur dessin :   
R : Rouge +  
F : Rouge -  
T : Vert +  
G : Vert -  
Y : Bleu +  
H : Bleu –

Couleur fond :   
U : Rouge +  
J : Rouge -  
I : Vert +  
K : Vert -  
O : Bleu +  
L : Bleu -

#### Autres :

Espace : Réinitialise la constante parmi 6 complexes et les couleurs de manière aléatoire  
A : Ajoute/Enlève l’écriture du frame rate et de la valeur actuelle de la constante complexe à gauche de l’image.  
Echap : Quitte l’application