UBFC

UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UFR de Sciences et Techniques

Master Pro Image et Intelligence Artificielle

Projet tuteuré

Implémentation de modèles fractales en CUDA

- **Encadreur**

  Christian Gentil

- **Préparé par**

  Nom : CHEREF

  Prénom : Mehdi

  Etudiant N° : 19 01 29 79

Sommaire

Liste des tables

Liste des figures

Contexte : Les algorithmes de génération de formes fractales sont la plupart du temps récursifs. Ils génèrent un nombre important d'éléments graphiques, et leur affichage en temps réel est souvent un défi (voir figure 1).

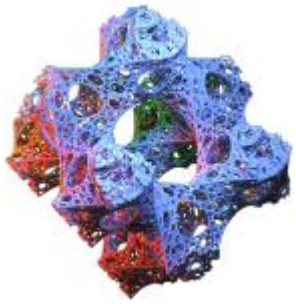


Figure 1 « Fractale complexe »

Sujet : Les systèmes itérés de fonctions est un outil pour générer très simplement des structures fractales complexes (objets de topologie et géométrie fractales). L'algorithme de visualisation est simple et massivement parallélisable. Nous proposons de réaliser une implémentation parallélisée sur GPU de cet algorithme en utilisant le langage de programmation CUDA.

1. Initiation aux systèmes itérés de fonctions, et implémentation de l'algorithme d'affichage standard en C++.
2. Etudier le langage de programmation CUDA puis porter l'algorithme sur GPU via CUDA.
3. Généralisation de l'algorithme aux systèmes itérés de fonctions contrôlés.
4. Etudier les représentations mémoires de la carte graphique utilisée par CUDA pour transférer directement les informations (VAO, VBO) dans le pipeline graphique.
5. Faire une étude de performance entre chaque étape.

Outils :

- Langages : **CUDA, C++ et GSLS**.
- Environnement : Linux.

Livrable :

1. Ce rapport explicatif.
2. Codes sources commentés.

Caractéristiques du matériel utilisé pour la réalisation de ce projet.

Table 1 « Caractéristique du matériels utilisés »

Matériel	Caractéristiques	
Processeur	Intel® core™ i3-4005U CPU @ 1.70GHz	
Mémoire (RAM)	6.00 GB	
Système type	64-bit Operating System, x64 based processor	
GPU	NVEDIA GEFORCE 920M	
Mémoire vidéo (GPU)	2.00 GB	
CUDA Spécifications	CUDA Driver Version	10.1
	CUDA RunTime Version	9.2
	CUDA Capability Version	3.5
	Micro-architecture	Kepler
	2 Multiprocessors	192 CUDA Cores/MP
	Globale memory	2004 MBytes
	Constante memory	65536 Bytes
	Shared memory per block	49152 Bytes
	Max number of threads per multiprocessor	2048
	Max number of threads per block	1024

Pour trouver la version de CUDA compatible avec votre carte graphique NVIDIA :

https://www.geforce.com/hardware/technology/cuda/supported-gpus?page=1&field_gpu_type_value=all

https://en.wikipedia.org/wiki/CUDA

Acronymes :

GPU : Graphics Processing Unit
CPU : Central Processing Unit
SMs : Streaming Multiprocessors
SISD : Single Instruction, Single Data
SIMD : Single Instruction, Multiple Data
MIMD : Multiple Instruction, Multiple Data
MISD : Multiple Instruction, Single Data
PE : Processing Element
ALU : Arithmetic Logic Unit
GPGPU : General-Purpose Computing on Graphics Processing Units

I. Etape 1 : **IFS** « Iterated Function System »

1. Fractale

Le terme fractal du latin « *fractus* », qui signifie « cassé » a été définie par le mathématicien Benoit Mandelbrot en 1975. Dans son ouvrage fondateur « *The Fractal Geometry of Nature* », il définit une fractale comme « une forme géométrique grossière ou fragmentée qui peut être divisé en parties, dont chacune est (au moins approximativement) une copie de taille réduite de l'ensemble. " [1]

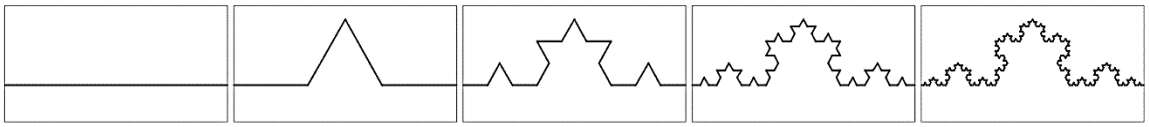


Figure I.1 « Célèbre motif fractal, découvert en 1904 par le mathématicien suédois Helge Von Koch » [1]

2. IFS « Iterated Function System »

Un outil pour générer très simplement des structures fractales complexes. [2]

$$T(F) = \bigcup_{i=1}^{N} Ti(F) \dots (1)$$

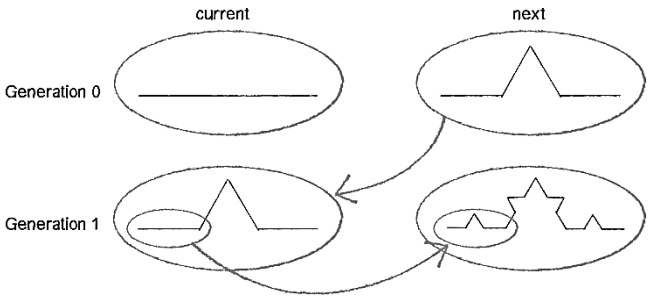


Figure I.2 « IFS systèmes itérés de fonctions » [1]

3. Implémentation de l'algorithme d'affichage sur CPU

On a implémenté trois différents modèles d'IFS sur CPU :

a. Le premier modèle est récursif, il calcule les adresses, applique les transformations géométriques puis il affiche le résultat final. Voir Algorithme 1.

b. Le deuxième modèle est Itératif, on a transformé le premier modèle récursif en une fonction itérative en utilisant une pile. Voir algorithme 2.

   Ce modèle est une implémentation de l'algorithme « *Depth-First* », qui a été étudié au 19ème siècle par le mathématicien français Charles Pierre Trémaux comme stratégie pour résoudre les labyrinthes. [3, 4]

c. Troisième modèle est aussi itératif, il trouve d'abord toutes les adresses possibles pour construire la fractale, puis il applique les transformations une par une et affiche le résultat. (L'inconvénient de ce modèle est qu'il refait beaucoup de multiplications matricielles couteuses). Voir Algorithme 3.

Note : pour les trois algorithmes suivants :

Soit **poly** la liste des vertex de la primitive
Soit **TransfList** la liste des transformations
Soit **iteration** le nombre d'itération

Algorithme 1 *: IFS_Recursif_CPU (poly, TransfList, iteration)*

```
Debut
    Si (iteration =0 )  Afficher le polygone
    Sinon
        Pourchaque Transformation dans TransfList Faire
            poly = Transformation × poly
            iteration= iteration-1
            IFS_Recursif_CPU(poly, TransfList, iteration)
        Fin
    Fin
Fin
```

Algorithme 2 *: IFS_Iteratif1_CPU (poly, TransfList, iteration)*

```
Pile pile
Debut
    TantQue (Vrai) Faire
        TantQue(iteration>0) Faire
            iteration= iteration-1
            i=taille(TransfList)
            TantQue(i>1) Faire
                empiler (pile, TransfList[i] × poly, iteration)
                i=i-1
            Fin
            poly = TransfList[1] × poly
        Fin
        Afficher le polygone
        Si (pile vide) sortir de la boucle
        Sinon
            poly, iteration = depiler(pile)
        Fin
    Fin
Fin
```

***Algorithme 3** : IFS_ Iteratif2_CPU (poly, TransfList, iteration)*

```
Debut
    Calculer les adresses
    Pourchaque Liste dans ListeAdresse Faire
        Pourchaque adresse dans Liste Faire
            Appliquer la transformation au polygone
        Fin
        Afficher le polygone
    Fin
Fin
```

- Voir l'étude de performance de ces trois algorithmes dans la partie V.

II. Etape 2 : GPU & CUDA

1. Introduction

Les progrès révolutionnaires réalisés par les ordinateurs basés sur GPU aident à accélérer les calculs analytiques, scientifiques et autres calculs intensifs. Grace à leur architecture massivement parallèle avec des milliers de cœurs, le GPU permet d'effectuer des tâches exigeantes en calcul beaucoup plus rapidement que les CPUs, car les CPUs ont un nombre de cœurs relativement petit. [5, 6]

2. Architecture des GPUs

Le GPU contient des centaines à des milliers d'unités de traitement arithmétique de même taille, c'est ce qui rend le GPU capable d'exécuter des milliers de threads simultanément. Pour s'exécuter simultanément, ces threads doivent être mutuellement indépendants sans aucun problème de synchronisation. Le parallélisme des threads dans un GPU convient pour exécuter le même programme sur des données différentes. [7]

Figure II.1 « CPU vs GPU »

L'architecture GPU se compose de deux composantes principale, la mémoire globale et les multiprocesseurs de streaming SMs. La mémoire globale est la mémoire principale du GPU et elle est accessible à la fois par le GPU et le CPU avec une bande passante

élevée. Alors que les SMs contiennent de nombreux cœurs simples qui exécutent les calculs parallèles, le nombre des SMs dans un périphérique et le nombre de cœurs dans les SMs diffèrent d'un périphérique à l'autre. [8]

3. Taxonomie de Flynn

La taxonomie de Flynn est une classification des différents types de parallélisme. Il identifie quatre classes d'architectures selon leurs instructions et leurs flux de données [9] :

- **SISD** (Single Instruction, Single Data) fait référence à l'architecture traditionnelle de von Neumann où un seul élément de traitement séquentiel (PE) fonctionne sur un seul flux de données.
- **SIMD** (Single Instruction, Multiple Data) effectue la même opération sur plusieurs éléments de données simultanément.
- **MIMD** (Multiple Instruction, Multiple Data) utilise plusieurs PE pour exécuter différentes instructions sur différents flux de données.
- **MISD** (Multiple Instruction, Single Data) utilise plusieurs PE pour exécuter différentes instructions sur un seul flux de données.

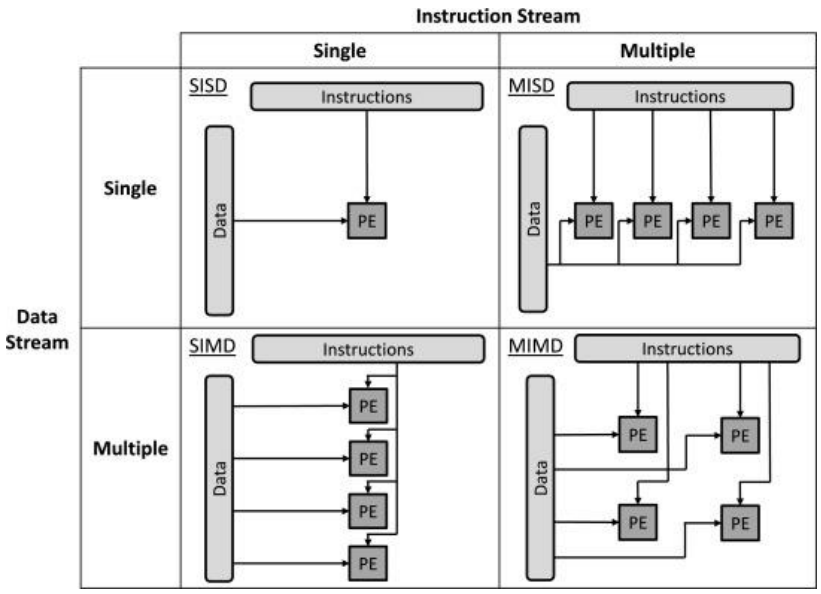


Figure II.2 « Taxonomie de Flynn »

4. CUDA

CUDA est une technologie de GPGPU (*General-Purpose Computing on Graphics Processing Units*) c'est-à-dire utilisant un processeur graphique (GPU) de NVIDIA pour exécuter des calculs généraux à la place du processeur (CPU). [10]

5. L'architecture de parallélisme et de la mémoire de CUDA

La figure **II.1** illustre cette architecture : [11]

À gauche, on voit comment les grilles et les blocs sont organisés dans CUDA. À droite, la hiérarchie des différents niveaux de mémoire sur GPU.

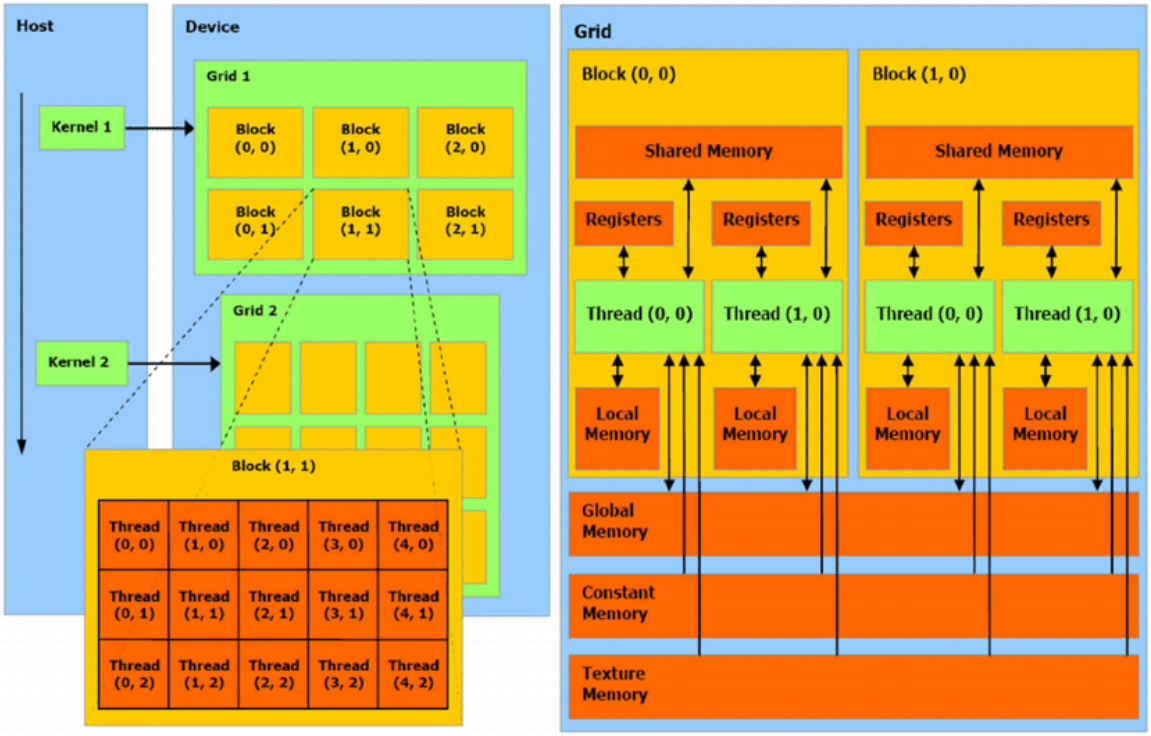


Figure II.3 « Schéma de l'architecture de parallélisme de CUDA et de son architecture de mémoire »

6. Les mémoires utilisées

Le choix des mémoires à un lien directe avec le besoin d'utilisation, c'est-à-dire les données d'entrées et de sorties d'une fonction écrite en langage CUDA, dans notre cas nous avons des données d'entrées constantes, comme les matrices de transformation et les coordonnées du polygone, et des données en sortie qui pourraient être le tableau contenant tous les vertex calculés de la fractale.

7. Mémoire Constante

On utilise la mémoire constante afin de stocker les données constantes pour les raisons suivantes : [12]

1. Les données ne vont pas changer au cours de l'exécution.
2. Tous les threads vont accéder à ces données à partir du même pointeur.

Avantage de l'utilisation de la mémoire constante :

- La lecture à partir de la mémoire constante est aussi rapide que la lecture à partir d'un registre.

```
const float h_tl[sizeTL] = {1.0, 0.5, 0.5,
                            0.0, 0.5, 0.0,
                            0.0, 0.0, 0.5,//T0
                            0.5, 0.0, 0.0,
                            0.5, 1.0, 0.5,
                            0.0, 0.0, 0.5,//T1
                            0.5, 0.0, 0.0,
                            0.0, 0.5, 0.0,
                            0.5, 0.5, 1.0 //T2
                            };

__constant__ float d_tl[sizeTL];//device transformation list

cudaMemcpyToSymbol(d_tl, h_tl, sizeof(float)*sizeTL);
```

Figure II.4 « Exemple d'utilisation de la mémoire constante »

La figure II.2 illustre comment utiliser la mémoire constante, on déclare d'abord la variable en utilisant le mot clé « __constant__ », et l'initialisation se fait avec la fonction « *cudaMemcpySymbol* ».

8. **Mémoire Partagée et mémoire Globale**

   Les threads d'un même bloc ont deux manières principales de communiquer des données entre elles. Le moyen le plus rapide serait d'utiliser la mémoire partagée. Cependant cette mémoire est très limitée en termes de stockage (16Ko uniquement), de plus, un thread ne peut pas accéder à une MP d'un autre block. En outre, la mémoire globale à une grande capacité de stockage et accessible par tous les threads. Pour la plupart des cartes vidéo vendues aujourd'hui, le GPU peut accéder au moins à 128 Mo de mémoire. [13]

9. **Mémoire Texture**

   Utiliser dans les pipelines de rendu OpenGL et DirectX classiques. [14]

III. Etape 3 : Implémentation des algorithmes IFS EN CUDA

Le plus grand défi était d'apporter les fonctions IFS (vu dans l'étape I) en CUDA, car en programmation parallèle (sur GPU) la notion de dépendance est interdite alors qu'il existe une forme de dépendance dans nos fonctions IFS sur CPU.

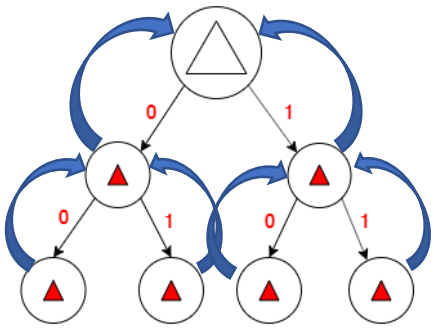


Figure III.1 « Dépendance des IFS »

1. Premier algorithme en CUDA

La première implémentation d'algorithme en CUDA était basée sur la même idée que le troisième IFS implémenté en CPU, c'est-à-dire qu'en premier lieu on calcule toutes les adresses possibles.

Le CUDA kernel utiliser pour calculer les adresses est le suivant :

```
__global__ void fillByLevel(short* result, size_t dim, size_t level, size_t fill_size, size_t fill_offset, size_t global_offset)
{
    if (level <= 0) return;

    short value = threadIdx.x;
    size_t start = fill_size * value + fill_offset + global_offset;
    size_t end = start + fill_size;
    for (size_t i = start; i < end; i++) result[i] = value;
    fillByLevel <<<1, dim >> > (result, dim, level - 1, fill_size / dim, start, global_offset);
    __syncthreads();
}
```

Figure III.2 « CUDA kernel utilisant la parallélisation dynamique »

Ce kernel récursif (« récursif » ne signifie pas « dépendant », dans CUDA chaque kernel est exécuté dans un seul thread indépendamment des autres kernels) est inspiré de la programmation parallèle dynamique CUDA. [15]

L'étude de performance de cet algorithme avec les trois algorithme IFS sur CPU (voir partie V), a montré que ce Kernel est beaucoup plus couteux, en termes de temps et de ressource. Ce qui nous a poussé à chercher une autre solution.

2. Deuxième Algorithme en CUDA

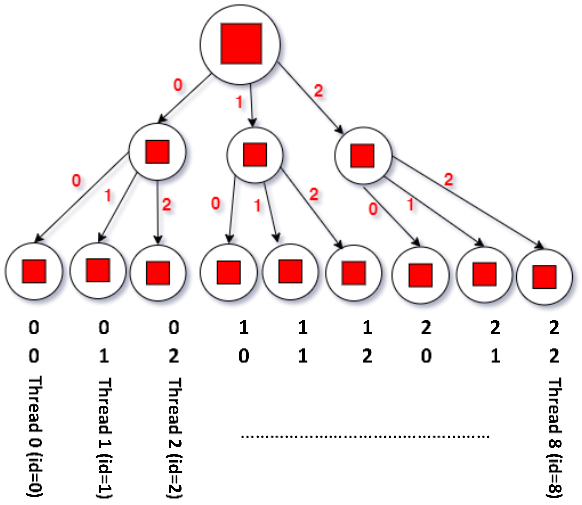
Le deuxième algorithme sur GPU permet de calculer les adresses mais aussi d'appliquer les transformations géométriques en même temps. L'algorithme est le suivant :

Algorithme 4 *: IFSCuda(VBO, I, B, P, LT)*

*/*VBO est un pointeur vers le vertex buffer */*
*/*soit I le nombre d'itération et B le nombre de transformation*/*
*/*soit LT la liste des transformations et P la primitive */*
/ chaque thread exécute une instance de cette fonction */*
/ chaque thread possède un identifiant* ***id*** ϵ $[0, B^I[$ **/*
Entier *: i = I-1*
Entier *: n = thread.id*
Debut
    TantQue *(i>=0)* ***do***
        t = n/B^i ;
        *P= LT[t]*P ; /* Appliquer la transformation */*
        n = n% B^i ;
        i = i-1 ;
    Fin
    Ajouter P au VBO
Fin

La complexité de l'algorithme 4 est de l'ordre O(n×m). Où n est le nombre d'itération et m est le nombre des vertex de la primitive.

Exemple d'application de l'algorithme 4 :
Soit B=3, I =2 le nombre total des thread N= B^I = 9, initialement on a i= I-1 et n=Thread.id



Thread **id**=3			
i	**n**	**B^i**	**t=n/B^i**
1	3	3^1=3	**1**
0	0	1	**0**
Thread **id**=8			
i	**n**	**B^i**	**t=n/B^i**
1	8	3^1=3	**2**
0	2	1	**2**

Figure III.3 « Exemple d'application de l'algorithme 4 »

La figure III.4 montre l'implémentation de l'algorithme 4 dans CUDA :

```
__global__ void IFSkernel(float *ver,short level,unsigned short dim, unsigned int Bi,size_t offset){
    size_t N=threadIdx.x + blockIdx.x * blockDim.x + offset;
    size_t n=N;
    unsigned short T;
    float *poly=new float[d_sizeV];
    float *p=new float[d_sizeV];
    memcpy(p, d_v, sizeof(float)*d_sizeV);
    short nbrVertex=d_sizeV/3;

    while(level>=0){
        T=n/Bi;

        for (short r = 0; r < nbrVertex ; r++)
        {
            for (short c = 0; c < 3 ; c++)
            {
                poly[r*3+c]=0;
                for (short k = 0; k < 3 ; k++)
                    poly[r*3+c]+=d_tl[d_offsetT[T]+r*3+k] * p[k*3+c];
            }
        }

        n=n%Bi;
        Bi=Bi/dim;
        level--;
        memcpy(p, poly, sizeof(float)*d_sizeV);
    }

    for(short i=0;i<3;i++)
        for(short j=0;j<3;j++) ver[N*d_sizeV+i*3+j]=poly[i+j*3];
}
```

Figure III.4 « Implémentation de l'algorithme 4 en CUDA »

3. **Gestion des block/thread par block** :
   La gestion des blocks et des threads par block est très importante et reste sujet à étude selon les caractéristiques de la carte graphique utilisée (voir Table 1), dans notre cas les caractéristiques liées à cette partie sont les suivant :

Table III.1 « Caractéristiques carte graphique lié au Multiprocesseur, block et thread »

2 Multiprocessors	192 CUDA Cores/MP
Max number of threads per multiprocessor	2048
Max number of threads per block	1024

Le nombre maximal de thread dans un block est donc 1024, maintenant comment on fait appel à notre kernel avec le bon nombre de <<< block, thread par block >>> en fonction du nombre totale des threads qui doivent être lancés dans l'IFS GPU ? nous proposons l'algorithme 5 suivant pour gérer cette tâche :

Algorithme 5 *: gestionBlock(I, B)*

```
/*soit I le nombre d'itération et B le nombre de transformation*/
Constante : maxTreadParBlock= 1024
Entier : totalThread= B^I
Begin
    if(totalThread<= maxTreadParBlock) then
        nbrBlock=1 ;
        threadParBlock= totalThread ;
        mode=0 ;
    else
        threadParBlock = maxTreadParBlock ;
        nbrBlock= totalThread / maxTreadParBlock ;
        mode = totalThread % maxTreadParBlock ;
    end
    ifsKernel<<< nbrBlock, threadParBlock>>>(…)
    if(mode !=0) then
        ifsKernel<<< 1, mode >>>(…)
    end

End
```

IV. Etape 4 : VBO et CUDA

Lors de l'implémentation de l'algorithme 4, et afin d'éviter le « *copy Memory From Device To Host* » et gagner plus de temps, nous avons donné les droits nécessaires à la fonction « ifsKernel » pour lui permettre de rajouter les vertex calculés directement dans le VBO, la figure IV.1 montre le bout de code qui donne ces droits.

```
struct cudaGraphicsResource *cuda_vbo_resource;
GLuint points_vbo;
float* d_vbo_ptr = 0;
```

```
//initialize a VBO
points_vbo = 0;
// generate 1 VBO buffer
glGenBuffers(1, &points_vbo);
// bind points_vbo to GL_ARRAY_BUFFER.
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
// locate the memory without initialize the values
glBufferData(GL_ARRAY_BUFFER, threads*9 * sizeof(float), 0, GL_DYNAMIC_DRAW);
```

```
//connet cuda_vbo_resource to points_vbo
cudaGraphicsGLRegisterBuffer(&cuda_vbo_resource, points_vbo, cudaGraphicsMapFlagsNone);
//give access authority of points_vbo to cuda
cudaGraphicsMapResources(1, &cuda_vbo_resource, 0);

size_t num_bytes;
//"verteses" points to the GPU memory data store of VBO (points_vbo) maped by cuda_vbo_resource
cudaGraphicsResourceGetMappedPointer((void **)&d_vbo_ptr, &num_bytes, cuda_vbo_resource);

IFSkernel<<<block,threadPerblock >>> (d_vbo_ptr,level-1,dim,Bi,0);
```

Figure IV.1 « VBO & CUDA »

V. Etape 5 : Etude de performance

Première étude de performance est élaborée sur les trois programmes ifs sur CPU (vu dans l'étape I), les trois algorithmes appliquent 3 transformations géométriques sur un triangle sur plusieurs niveaux/itérations pour construire le triangle de *Sierpinski*, les résultats sont présentés dans la table V.1 :

Table V.1 « Temps d'exécution des trois programmes ifs sur CPU »

Itération / Temps	5	8	12	15
IFS CPU Récursif	0.002816 s	0.038799 s	1.96780 s	45.2063 s
IFS CPU Itératif 1	0.000610 s	0.020954 s	1.39534 s	33.5501 s
IFS CPU Itératif 2	0.003839 s	0.055317 s	5.52907 s	178.611 s

L'algorithme itératif 1 est le plus performant, ce qui est très logique car l'algorithme récursif gère **en plus** les appelle récursif en utilisant des piles, le deuxième algorithme itératif refait des multiplications matricielles couteuses.

La figure V.1 montre la fractale (le triangle de *Sierpinski*) de niveau 5 calculé sur CPU.

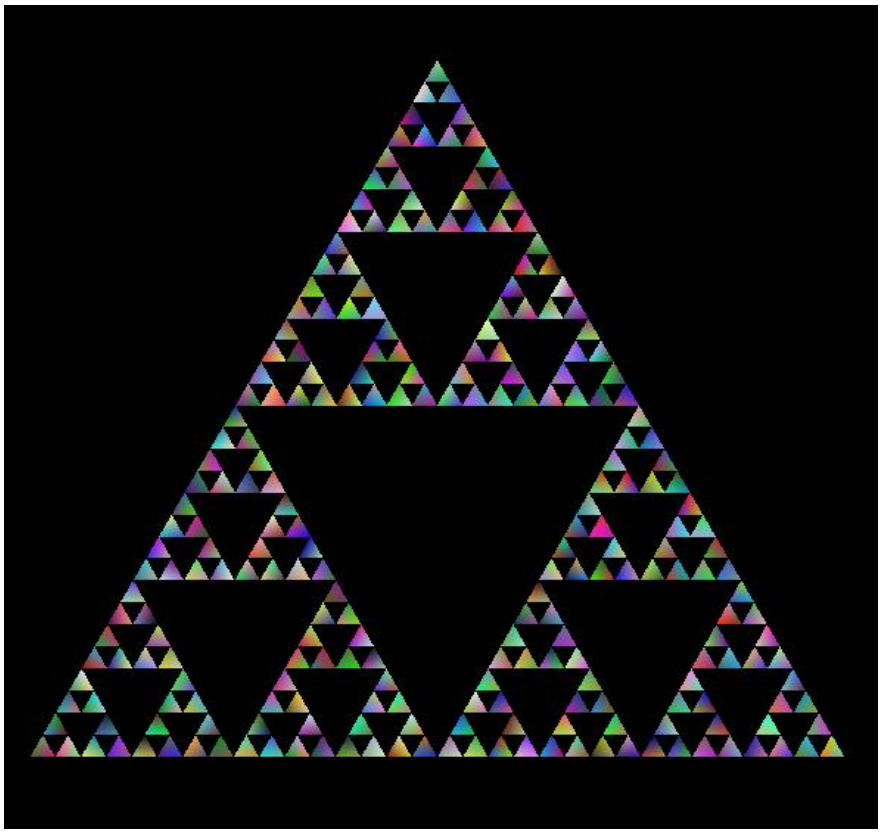


Figure V.1 « Résultat triangle de *Sierpinski calculé avec IFS CPU* »

La deuxième étude de performance est élaborée sur tous les algorithmes implémentés sur CPU ainsi que GPU, ces algorithmes appliquent trois transformations géométriques sur un triangle sur plusieurs niveaux/itérations pour construire le triangle de *Sierpinski*, les résultats sont présentés dans la table V.2 :

Table V.2 « Temps d'exécution de tous les programmes ifs CPU & GPU »

Itération / Temps	5	8	**11**	
IFS CPU algo 1	0.002816 s	0.038799 s	1.96780 s	**≈ T × 14**
IFS CPU algo 2	0.000610 s	0.020954 s	1.39534 s	**≈ T × 10**
IFS CPU algo 3	0.003839 s	0.055317 s	5.52907 s	**≈ T × 39**
IFS GPU 1	0.127290 s	0.262831 s	4.15053 s	**≈ T × 28**
IFS GPU 2 (algo 4)	0.138175 s	0.141092 s	**T= 0.13952 s**	

Pour des raisons de performance matériel (carte graphique pas trop puissante), nous n'avons pas pu tester les algorithmes GPU sur des niveaux/itérations supérieurs à 11.

D'après les résultats de la table V.2, on remarque que le temps d'exécution de l'IFS GPU 2 (algorithme 4) reste stable comparé aux autres algorithmes. Sachant que la complexité de l'algorithme 4 est de l'ordre **O(n×m)** (**n** le niveau de la fractale et **m** le nombre de vertex de la primitive), on peut dire qu'une grande partie du temps d'exécution est consacré à l'initialisation des threads dans CUDA, et que la performance de cet algorithme sera encore plus importante dans des niveaux supérieurs à 11.

VI. Conclusion

Dans ce projet nous avons réalisé un Système de Fonctions Itérés sur GPU en utilisant le langage de programmation CUDA.

La complexité de l'algorithme 4 est de l'ordre O(n×m), on a réussi à réduire la complexité des IFS CPU qui est de l'ordre $O(B^n \times m)$.

L'exécution de l'algorithme 4 GPU proposé peut être 39 fois plus rapide que certains Systèmes de Fonctions Itérés CPU.

Il existe plusieurs recherches scientifiques sur l'implémentation parallèle de l'algorithme « Depth-First » [16, 17], et l'algorithme 4 peut être adapté pour paralléliser le « Depth-First » pour les arbres réguliers.

Les résultats des performances sont prometteurs et peuvent enrichir la littérature.

VII. Perspective

- Proposer un autre algorithme de gestion des block/Thread par block.
- Améliorer l'algorithme 4 afin de diminuer le nombre total des threads.
- Utiliser un tableau d'index pour éliminer les vertex doublés dans le VBO.
- Réaliser des tests sur une carte graphique NVIDIA plus puissante.

VIII. Bibliographie

[1] : Daniel Shiffman, 'The nature of code', (2012) <https://natureofcode.com/book/chapter-8-fractals/ > Consulté le : 06.03.20

[2] : Cours de Géraldine Morin* et Christian Gentil**, *VORTEX - IRIT-ENSEENSEEIHT, LE2I-Université de Bourgogne, novembre 2010.

[3] : Even, Shimon (2011), Graph Algorithms (2nd ed.), Cambridge University Press, pp. 46–48.

[4] : Sedgewick, Robert (2002), Algorithms in C++: Graph Algorithms (3rd ed.), Pearson Education.

[5] : Yuen DA, et al. GPU Solutions to Multi-scale Problems in Science and Engineering. Springer; 2013. http://www.springer.com/us/book/9783642164040

[6] : Zahran M. Graphics Processing Units (GPUs): Architecture and Programming (Multi-GPU Systems) – Lecture. [cited January 9, 2017] [Internet]. Disponible ici : http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture9.pdf

[7] : An Introduction to Modern GPU Architecture, Ashu Rege, Director of Developer Technology NVIDIA. [cited December 31, 2016].
http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

[8] : Abdelrahman Ahmed Mohamed Osman, GPU Computing Taxonomy, Published: July 19th 2017, DOI: 10.5772/intechopen.68179, https://www.intechopen.com/books/recent-progress-in-parallel-and-distributed-computing/gpu-computing-taxonomy

[9] : Parallel Programming Concepts and Practice, Chapter 3 - Modern Architectures, 2018, Pages 47-75, https://www.sciencedirect.com/topics/computer-science/single-instruction-single-data

[10] : Anand Lal Shimpi et Wilson, Derek, « Nvidia's GeForce 8800 (G80): GPUs Re-architected for DirectX 10 » Consulté le : 06.03.20

[11] : Xuelin Cui et Hongbin Guo, conference June 2017, Stencil-based Discrete Gradient Transform Using GPU Device in Compressed Sensing MRI <
https://www.researchgate.net/publication/317601778_Stencil-based_Discrete_Gradient_Transform_Using_GPU_Device_in_Compressed_Sensing_MRI >

[12] : Whisky & Rum bestellen, CUDA Programming 2012, < http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html > Consulté le : 06.03.20

[13] : Whisky & Rum bestellen, CUDA Programming 2012, < http://cuda-programming.blogspot.com/2013/01/shared-memory-and-synchronization-in.html > Consulté le : 06.03.20

[14] : Whisky & Rum bestellen, CUDA Programming 2012, < http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html > Consulté le : 06.03.20

[15] : NVIDIA Corporation, 2012,
<http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf >
Consulté le : 07.03.20

[16] : Parallel Depth First on GPU, M. Naumov, A. Vrielink and M. Garland, GTC 2017,
< http://on-demand.gputechconf.com/gtc/2017/presentation/s7469-maxim-naumov-parallel-depth-first-on-gpu.pdf >
Consulté le : 08.03.20

[17] : Maxim Naumov, Alysson Vrielink, Michael Garland, Parallel Depth-First Search for Directed Acyclic Graphs, DOI : < https://doi.org/10.1145/3149704.3149764 >