# Sommaire :

1. Résultats attendus
2. Démarche générale
3. Prérequis
4. CUDA : création du programme de calcul avec plusieurs GPU
5. PYTHON 1 : Création des images taille réel avec plusieurs GPU
6. PYTHON 2 : Création DZI
7. WEB : Création site WEB
8. Remerciements

# Résultats Attendus

On chercher à obtenir des fractale de Julia en deux formats d’image avec une taille importante (>32k px de coté) :

* Bi couleur (N&B)
* En nuance de gris

Figure : Image bi-couleurs de la fractale de Julia

Figure : Image en nuance de gris de la fractale de Julia

De même on, utilisera un site web locale ou sur réseaux pour visualiser les fractales de Julia, l’interface se tel que :

Figure : Interface WEB

Description de l’interface WEB :

1. Titre Dynamique avec les valeurs de X et Y
2. Axe des X : permet de modifier la valeur de X
3. Axe des Y : permet de modifier la valeur de Y
4. Option d’affichage et bouton de téléchargement de l’image d’origine
5. Explorateur de la fractale, avec un zoom important possible.

# Démarche générale

Il y a 4 étapes à respecte :

## 1. Création d’un tableau du nombre d’itération de chaque pixel de l’image

- Outil : CUDA (C / C++)  
  
- OS : Linux (WSL 2 Ubuntu)  
  
- Matériel : Carte graphique Nvdia 4 Go RAM

## 2. Transformation du tableau du nombre d’itération en images et compression du tableau pour optimise l’usage du disque dur.

- Outil : CUDA (C / C++) et python 3  
  
- OS : Linux (WSL 2 Ubuntu)  
  
- Matériel : Carte graphique Nvdia 4 Go RAM

## 3. Création d’image zoomable avec le logiciel « openseadragon » et « deepzoom.py »

- Outil : python 3  
  
- OS : Linux (WSL 2 Ubuntu) ou Windows

## 4. Création du site web pour visualiser les fractales

- Outil : python 3 / HTML / JS  
  
- OS : Linux (WSL 2 Ubuntu) ou Windows

# 3. Prérequis

## 1. Activer WSL2 et NVIDIA

<https://learn.microsoft.com/fr-fr/windows/ai/directml/gpu-cuda-in-wsl>

<https://docs.nvidia.com/cuda/wsl-user-guide/index.html>

# Installation des Pilotes et Toolkits NVIDIA  
sudo apt-key del 7fa2af80  
wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86\_64/cuda-keyring\_1.1-1\_all.deb  
sudo dpkg -i cuda-keyring\_1.1-1\_all.deb  
sudo apt-get update  
sudo apt-get -y install cuda-toolkit-12-4  
sudo apt-get -y install cuda-tools-12-4  
sudo apt-get -y install cuda-runtime-12-4  
sudo apt-get -y install cuda-12-4  
  
# Installation pip3  
sudo apt install python3-pip  
  
# Installation des Librairies Python  
pip3 install numpy numba pillow joblib py7zr

# 4. CUDA : création du programme de calcul avec plusieurs GPU

Le code cuda permet d’utiliser les GPU NVDIA comme centre de calculs.

Le code que je propose est décompose en 5 parties :

## 1. Le header

C’est le code commun entre le code cuda et c++, on y trouve :

* Le type de fractale à générer : Type\_Fractal

// Définition de l'énumération pour le type de fractale  
enum Type\_Fractal { Mandelbrot, Julia };

* La structure **Complex** pour représenter les nombres complexes

// Définition de la structure Complex pour représenter les nombres complexes  
struct Complex  
{  
 double x, y; // Partie réelle et imaginaire  
  
 // Constructeur pour initialiser un nombre complexe  
 \_\_host\_\_ \_\_device\_\_  
 Complex(double a = 0.0, double b = 0.0) : x(a), y(b) {}  
  
 // Surcharge de l'opérateur + pour l'addition de deux nombres complexes  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator+(const Complex &other) const  
 {  
 return Complex(x + other.x, y + other.y);  
 }  
  
 // Surcharge de l'opérateur - pour la soustraction de deux nombres complexes  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator-(const Complex &other) const  
 {  
 return Complex(x - other.x, y - other.y);  
 }  
  
 // Surcharge de l'opérateur \* pour la multiplication de deux nombres complexes  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator\*(const Complex &other) const  
 {  
 return Complex(x \* other.x - y \* other.y, x \* other.y + y \* other.x);  
 }  
  
 // Fonction pour calculer la norme d'un nombre complexe  
 \_\_host\_\_ \_\_device\_\_ double norm() const  
 {  
 return sqrt(x \* x + y \* y);  
 }  
  
 // Fonction pour élever un nombre complexe à une puissance donnée  
 \_\_host\_\_ \_\_device\_\_  
 Complex power(double p) const  
 {  
 double radius = sqrt(x \* x + y \* y);  
 double angle = atan2(y, x);  
 double radius\_p = pow(radius, p);  
 double angle\_p = p \* angle;  
  
 return Complex(radius\_p \* cos(angle\_p), radius\_p \* sin(angle\_p));  
 }  
};

* La structure **ParameterPicture** pour stocker les paramètres de l’image fractale

// Définition de la structure ParameterPicture pour stocker les paramètres de l'image fractale  
struct ParameterPicture  
{  
 long lenG; // Longueur globale en 3D  
 long lenL; // Longueur locale en 2D  
 double2 start; // Point de départ de l'image  
 double size; // Taille d'un côté de l'image  
 Type\_Fractal type\_fractal; // Type de fractale (Mandelbrot ou Julia)  
 double2 coef\_julia; // Coefficients pour la fractale de Julia  
 double power\_value; // Valeur de la puissance  
 long iter\_max; // Nombre maximal d'itérations  
 long id; // Identifiant de l'image  
  
 // Constructeur pour initialiser un objet ParameterPicture  
 \_\_host\_\_ \_\_device\_\_ ParameterPicture(long id, long lenG, double2 start, double size, double power\_value, long iter\_max, Type\_Fractal type\_fractal, double2 coef\_julia = make\_double2(0.0, 0.0))   
 : id(id), power\_value(power\_value), iter\_max(iter\_max), type\_fractal(type\_fractal), coef\_julia(coef\_julia), lenG(lenG), lenL(floorf(sqrtf((float)lenG))), start(start), size(size) {};  
  
 // Fonction pour obtenir la taille de l'image en 3D  
 \_\_host\_\_ \_\_device\_\_ size\_t Get\_size\_array\_3D() const  
 {  
 return (size\_t)lenG \* (size\_t)lenG \* (size\_t)lenG;  
 }  
  
 // Fonction pour obtenir la taille de l'image en 2D  
 \_\_host\_\_ \_\_device\_\_ size\_t Get\_size\_array\_2D() const  
 {  
 return (size\_t)lenG \* (size\_t)lenG \* (size\_t)lenL \* (size\_t)lenL;  
 }  
  
 // Fonction pour obtenir la position en coordonnées double dans l'image  
 \_\_host\_\_ \_\_device\_\_ double2 GetPose\_double(int x, int y, int z) const  
 {  
 int id = 0;  
 for (long x\_ = 0; x\_ < lenL; x\_++)  
 {  
 for (long y\_ = 0; y\_ < lenL; y\_++)  
 {  
 if (id == z)  
 {  
 return make\_double2(start.x + ((double)x\_ \* size) + ((double)x / (double)lenG \* size), start.y + ((double)y\_ \* size) + ((double)y / (double)lenG \* size));  
 }  
 id++;  
 }  
 }  
 return make\_double2(0.0, 0.0);  
 }  
  
 // Fonction pour obtenir la position en coordonnées long dans l'image  
 \_\_host\_\_ \_\_device\_\_ long2 GetPose\_long(int x, int y, int z) const  
 {  
 int id = 0;  
 for (long x\_ = 0; x\_ < lenL; x\_++)  
 {  
 for (long y\_ = 0; y\_ < lenL; y\_++)  
 {  
 if (id == z)  
 {  
 return make\_long2((x\_ \* lenG) + (long)x, (y\_ \* lenG) + (long)y);  
 }  
 id++;  
 }  
 }  
 return make\_long2(0, 0);  
 }  
  
 // Fonction pour obtenir l'index 3D d'une position dans l'image  
 \_\_host\_\_ \_\_device\_\_ long Get\_index\_3D(int x, int y, int z) const  
 {  
 if (x < 0 || (long)x >= lenG)  
 return -1;  
 if (y < 0 || (long)y >= lenG)  
 return -1;  
 if (z < 0 || (long)z >= lenL \* lenL)  
 return -1;  
  
 return (long)z \* lenG \* lenG + (long)y \* lenG + (long)x;  
 }  
  
 // Fonction pour obtenir l'index 2D d'une position dans l'image  
 \_\_host\_\_ \_\_device\_\_ long Get\_index\_2D(int x, int y, int z) const  
 {  
 if (x < 0 || (long)x >= lenG)  
 return -1;  
 if (y < 0 || (long)y >= lenG)  
 return -1;  
 if (z < 0 || (long)z >= (lenL \* lenL))  
 return -1;  
  
 long2 pose = GetPose\_long(x, y, z);  
 return pose.y \* lenG \* lenL + pose.x;  
 }  
  
 // Fonction pour définir une valeur dans les données de l'image à une position donnée  
 \_\_host\_\_ \_\_device\_\_ void Set\_Value(int x, int y, int z, long \*data, long value) const  
 {  
 long index = Get\_index\_2D(x, y, z);  
 if (index >= 0)  
 {  
 data[index] = value;  
 }  
 }  
  
 // Fonction pour obtenir une valeur des données de l'image à une position donnée  
 \_\_host\_\_ \_\_device\_\_ long Get\_Value(int x, int y, int z, long \*data) const  
 {  
 long index = Get\_index\_2D(x, y, z);  
 if (index >= 0)  
 {  
 return data[index];  
 }  
 else  
 {  
 return 0;  
 }  
 }  
  
 // Fonction pour imprimer les paramètres de l'image dans un fichier  
 \_\_host\_\_ void print\_file(std::string path\_file) const  
 {  
 std::ofstream myfile;  
 myfile.open(path\_file, std::ios::app);  
 myfile << "id = " << id << std::endl;  
  
 myfile << "lenG = " << lenG << std::endl;  
 myfile << "lenL = " << lenL << std::endl;  
  
 myfile << "start\_x = " << start.x << std::endl;  
 myfile << "start\_y = " << start.y << std::endl;  
  
 myfile << "size = " << size << std::endl;  
 myfile << "type\_fractal = " << type\_fractal << std::endl;  
 myfile << "coef\_julia\_x = " << coef\_julia.x << std::endl;  
 myfile << "coef\_julia\_y = " << coef\_julia.y << std::endl;  
  
 myfile << "power\_value = " << power\_value << std::endl;  
 myfile << "iter\_max = " << iter\_max << std::endl;  
 myfile.close();  
 }  
};

## 2. Le code cuda

C’est le code qui calcul la fractale de Julia ou de Mandelbrot, on y trouve :

* Kernel\_Picture : Kernel CUDA pour générer une image fractale

// Kernel CUDA pour générer une image fractale  
\_\_global\_\_ void Kernel\_Picture(ParameterPicture parameter\_picture, long \*data)  
{  
 // Calcul des indices 3D pour chaque thread  
 int idx = blockIdx.x \* blockDim.x + threadIdx.x;  
 int idy = blockIdx.y \* blockDim.y + threadIdx.y;  
 int idz = blockIdx.z \* blockDim.z + threadIdx.z;  
  
 // Obtenir l'index 2D correspondant  
 long index = parameter\_picture.Get\_index\_2D(idx, idy, idz);  
  
 // Si l'index est valide  
 if (index >= 0)  
 {  
 // Obtenir la position complexe correspondante  
 double2 pos\_double = parameter\_picture.GetPose\_double(idx, idy, idz);  
 Complex z(pos\_double.x, pos\_double.y);  
 Complex c(pos\_double.x, pos\_double.y);  
  
 // Si le type de fractale est Julia, utiliser les coefficients de Julia  
 if (parameter\_picture.type\_fractal == Type\_Fractal::Julia)  
 {  
 c.x = parameter\_picture.coef\_julia.x;  
 c.y = parameter\_picture.coef\_julia.y;  
 }  
   
 long iter = 0;  
  
 // Calculer le nombre d'itérations pour la fractale  
 while (z.norm() < 2.0 && iter < parameter\_picture.iter\_max)  
 {  
 z = z.power(parameter\_picture.power\_value) + c;  
 iter++;  
 }  
  
 // Stocker le nombre d'itérations dans le tableau de données  
 data[index] = iter;  
 }  
}

* RUN : la fonction pour exécuter le kernel CUDA

// Fonction pour exécuter le kernel CUDA  
cudaError\_t RUN(ParameterPicture parameter\_picture, long \*datas, int id\_cuda)  
{  
 // Calculer la taille des données à allouer  
 size\_t size = parameter\_picture.Get\_size\_array\_2D() \* sizeof(long);  
 long \*dev\_datas = 0;  
 cudaError\_t cudaStatus;  
  
 // Définir la configuration des threads et des blocs  
 const dim3 threadsPerBlock(16, 16, 4);  
 const dim3 numBlocks((parameter\_picture.lenG + threadsPerBlock.x - 1) / threadsPerBlock.x,   
 (parameter\_picture.lenG + threadsPerBlock.y - 1) / threadsPerBlock.y,   
 (parameter\_picture.lenG + threadsPerBlock.z - 1) / threadsPerBlock.z);  
  
 // Sélectionner le GPU à utiliser  
 cudaStatus = cudaSetDevice(id\_cuda);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  
 goto Error;  
 }  
  
 // Allouer de la mémoire sur le GPU pour les données  
 cudaStatus = cudaMalloc((void \*\*)&dev\_datas, size);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaMalloc failed!");  
 goto Error;  
 }  
  
 // Lancer le kernel CUDA  
 Kernel\_Picture<<<numBlocks, threadsPerBlock>>>(parameter\_picture, dev\_datas);  
  
 // Vérifier si le lancement du kernel a échoué  
 cudaStatus = cudaGetLastError();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "Kernel\_Picture launch failed: %s\n", cudaGetErrorString(cudaStatus));  
 goto Error;  
 }  
  
 // Attendre la fin de l'exécution du kernel  
 cudaStatus = cudaDeviceSynchronize();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching Kernel\_Picture!\n", cudaStatus);  
 goto Error;  
 }  
  
 // Copier les données du GPU vers la mémoire de l'hôte  
 cudaStatus = cudaMemcpy(datas, dev\_datas, size, cudaMemcpyDeviceToHost);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaMemcpy failed!");  
 goto Error;  
 }  
  
 // Libérer la mémoire allouée sur le GPU  
 cudaFree(dev\_datas);  
  
 // Réinitialiser le GPU  
 cudaStatus = cudaDeviceReset();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaDeviceReset failed!");  
 return cudaStatus;  
 }  
  
 return cudaSuccess;  
  
Error:  
 // En cas d'erreur, libérer la mémoire allouée sur le GPU  
 cudaFree(dev\_datas);  
 return cudaStatus;  
}

## 3. Le code C++

C’est le code qui permet de gérer la création de fractales de Julia ou de Mandelbrot, on y trouve :

* File\_Generate : la structure pour gérer les fichiers (.bin et .txt)

// Structure pour générer des fichiers  
struct File\_Generate  
{  
 std::string bin, txt; // Chemins des fichiers binaires et texte  
 bool exist; // Indicateur si le fichier existe  
 File\_Generate(std::string bin, std::string txt) : bin(bin), txt(txt) {}  
};

* RUN : Déclaration de la fonction CUDA externe

// Déclaration de la fonction CUDA externe  
extern cudaError\_t RUN(ParameterPicture parameter\_picture, long \*datas, int id\_cuda);

* CreateFolder : Fonction pour créer le dossier de travail

// Fonction pour créer le dossier de travail  
std::string CreateFolder(std::string name, std::string dirBase)  
{  
 std::string dirNameBase = dirBase;  
 std::string dirName = dirNameBase + "/" + name;  
  
 mkdir(dirNameBase.c\_str(), 0777);  
 if (mkdir(dirName.c\_str(), 0777) == 0)  
 { // Note : 0777 donne les droits d'accès rwx pour tous  
 std::cout << "Directory created: " << dirName << std::endl;  
 }  
 else  
 {  
 std::cout << "Failed to create directory!" << std::endl;  
 }  
  
 return dirName;  
}

* if\_file\_exist : Fonction pour vérifier si un fichier existe

// Fonction pour vérifier si un fichier existe  
bool if\_file\_exist(const std::string &name)  
{  
 std::ifstream f(name.c\_str());  
 return f.good();  
}

* write\_bin : Fonction pour écrire des données binaires dans un fichier

// Fonction pour écrire des données binaires dans un fichier  
bool write\_bin(std::string path\_file, long \*data, size\_t size)  
{  
 std::ofstream outfile(path\_file, std::ios::out | std::ios::binary);  
 if (!outfile)  
 {  
 std::cerr << "Cannot open file for writing.\n";  
 return false;  
 }  
  
 outfile.write(reinterpret\_cast<char \*>(data), size \* sizeof(long));  
 outfile.close();  
  
 free(data);  
 return true;  
}

* run : Fonction supervision pour lancement de calculs d’une fractale

// Fonction supervision pour lancement de calculs d'une fractale   
File\_Generate run(ParameterPicture parameter\_picture, std::string baseDir, int id\_cuda)  
{  
 // Création des chemins des fichiers  
 std::string path\_dir = CreateFolder("id\_" + std::to\_string(parameter\_picture.id), baseDir);  
 std::string path\_txt = path\_dir + "/parameters.txt";  
 std::string path\_bin = path\_dir + "/data.bin";  
  
 // Initialisation de la structure File\_Generate  
 File\_Generate file\_generate(path\_bin, path\_txt);  
 file\_generate.exist = if\_file\_exist(path\_txt);  
  
 if (file\_generate.exist)  
 return file\_generate;  
  
 long \*datas = 0;  
 try  
 {  
 size\_t size = parameter\_picture.Get\_size\_array\_2D() \* sizeof(long);  
 datas = (long \*)malloc(size);  
 cudaError\_t cudaStatus;  
  
 cudaStatus = RUN(parameter\_picture, datas, id\_cuda);  
 if (cudaStatus == cudaSuccess)  
 {  
 write\_bin(path\_bin, datas, parameter\_picture.Get\_size\_array\_2D());  
 parameter\_picture.print\_file(path\_txt);  
 file\_generate.exist = true;  
 }  
 else  
 {  
 file\_generate.exist = false;  
 }  
 }  
 catch (const std::exception &)  
 {  
 free(datas);  
 file\_generate.exist = false;  
 if (if\_file\_exist(path\_txt))  
 std::remove(path\_txt.c\_str());  
 if (if\_file\_exist(path\_bin))  
 std::remove(path\_bin.c\_str());  
 }  
  
 return file\_generate;  
}

* Get\_nbfiles\_bin : Fonction pour obtenir le nombre de fichiers binaires existants

// Fonction pour obtenir le nombre de fichiers binaires existants  
int Get\_nbfiles\_bin(std::vector<File\_Generate> Files\_G)  
{  
 int count = 0;  
 for (File\_Generate &file : Files\_G)  
 {  
 if (file.exist)  
 {  
 file.exist = if\_file\_exist(file.bin);  
 if (file.exist)  
 count++;  
 }  
 }  
 return count;  
}

* Open\_file\_txt : Fonction pour ouvrir un fichier texte et lire son contenu

/ Fonction pour ouvrir un fichier texte et lire son contenu  
std::string Open\_file\_txt(std::string path\_file)  
{  
 std::string myText;  
 std::string out;  
 std::ifstream MyReadFile(path\_file);  
  
 while (getline(MyReadFile, myText))  
 {  
 out = myText;  
 std::cout << path\_file << " contient " << myText << std::endl;  
 }  
  
 MyReadFile.close();  
 return out;  
}

* Main : Fonction principale qui est exécuté au lancement

int main()  
{  
 //coté en pixel d'une tuile, il y a int(sqrt(lenG)) de tuile par coté  
 //exemple pour 720 ==> il y a int(sqrt(720)) = 26 tuiles donc 26\*720 = 18 720 px de coté soit une image de 350 438 400 px en tout  
 //donc un fichier binaire de 2 803 507 200 octes soit 2.8 Go.  
 const long lenG = 720;  
  
 // nombre de fichier binaire max non traité par le scripte python  
 const int max\_bin\_files = 4;  
  
 //Borne min max de X  
 const double coef\_x\_min = -1.5;  
 const double coef\_x\_max = 1.5;  
  
 //pas d'itération de X et Y  
 const double coef\_pas = 0.1;  
  
 // Vérification de l'existence du fichier id\_cuda.txt  
 std::string path\_file\_id\_cuda = "./parameters/id\_cuda.txt";  
 int id\_cuda = 0;  
 std::string id\_cuda\_str = "";  
 if (if\_file\_exist(path\_file\_id\_cuda))  
 {  
 id\_cuda\_str = Open\_file\_txt(path\_file\_id\_cuda);  
 id\_cuda = std::stoi(id\_cuda\_str);  
 }  
 else  
 {  
 std::cout << "file not existe " << path\_file\_id\_cuda << std::endl;  
 return 1;  
 }  
  
 // Vérification de l'existence du fichier min.txt  
 std::string path\_file\_min = "./parameters/min.txt";  
 double min\_value = 0.0;  
 if (if\_file\_exist(path\_file\_min))  
 {  
 std::string min\_str = Open\_file\_txt(path\_file\_min);  
 min\_value = std::stod(min\_str);  
 }  
 else  
 {  
 std::cout << "file not existe " << path\_file\_min << std::endl;  
 return 1;  
 }  
  
 // Vérification de l'existence du fichier max.txt  
 std::string path\_file\_max = "./parameters/max.txt";  
 double max\_value = 0.0;  
 if (if\_file\_exist(path\_file\_max))  
 {  
 std::string max\_str = Open\_file\_txt(path\_file\_max);  
 max\_value = std::stod(max\_str);  
 }  
 else  
 {  
 std::cout << "file not existe " << path\_file\_max << std::endl;  
 return 1;  
 }  
  
 std::vector<File\_Generate> Files\_G;  
  
 // Construction du nom de base du répertoire  
 std::string baseDir = "datas\_" + id\_cuda\_str + "\_" + std::to\_string(lenG) + "p";  
 long id = 0;  
  
 // Boucles pour générer des fichiers pour différentes valeurs de coef\_x et coef\_y  
 for (double coef\_x = coef\_x\_min ; coef\_x <= coef\_x\_max; coef\_x += coef\_pas)  
 {  
 for (double coef\_y = min\_value; coef\_y < max\_value; coef\_y += coef\_pas)  
 {  
 std::cout << "id = " << id << std::endl;  
 std::cout << "Get\_nbfiles\_bin " << Get\_nbfiles\_bin(Files\_G) << std::endl;  
  
 // Attente si le nombre de fichiers binaires existants dépasse la limite  
 while (Get\_nbfiles\_bin(Files\_G) >= max\_bin\_files)  
 {  
 std::cout << "Get\_nbfiles\_bin " << Get\_nbfiles\_bin(Files\_G) << std::endl;  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(60ll \* 1000ll));  
 }  
  
 id++;  
 ParameterPicture parameter\_picture(id, lenG, make\_double2(-2.0, -2.0), (2.0 \* 2.0) / (double)floorf(sqrtf((float)lenG)), 2, 2024, Type\_Fractal::Julia, make\_double2(coef\_x, coef\_y));  
 Files\_G.push\_back(run(parameter\_picture, baseDir, id\_cuda));  
 }  
 }  
}

## 4. Le scripte pour compiler le programme.

C’est le scripte qui permet de générer l’application

/usr/local/cuda/bin/nvcc -c src/main.cu -o bin/main.o -I/usr/local/cuda/lib64 -I/usr/local/cuda/extras/CUPTI/lib64  
g++ -c -I/usr/local/cuda/include src/main.cpp -o bin/main\_cpp.o   
g++ bin/main.o bin/main\_cpp.o -o main -lcudart -L/usr/local/cuda/lib64 -L/usr/local/cuda/extras/CUPTI/lib64

## 5. Les paramètres

C’est les paramètres de calculs externes au programme, on y trouve :

* L’id de la care nvdia à utiliser de 0 à N, n étant le nombre -1 de cartes graphiques disponibles
* La borne minimale du coef y de Julia
* La borne maximale du coef y de Julia

## Exécution du programme

Compilation de code depuis le répertoire **01\_Creation\_Datas** :

$ bash ./make\_main.sh

Exécution du code depuis le répertoire **01\_Creation\_Datas** :

$ ./main

# 4 PYTHON 1 : Création des images taille réel avec plusieurs GPU