

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

# Contents :

1. Expected results
2. General approach
3. Prerequisites
4. CUDA: creating a calculation program with several GPUs
5. PYTHON 1: Creating life-size images with multiple GPUs
6. PYTHON 2: DZI creation
7. WEB : WEB site creation

# 1. Expected results

Website reproduction: https:[//fractals-julia.com/](https://fractals-julia.com/)

We are looking to obtain Julia fractals in two image formats with a large size (>32k px sides):

* Bi color (B&W)
* In shades of grey

Figure: Two-colour image of the Julia fractal

Figure: Grayscale image of the Julia fractal

Similarly, we'll use a local or networked website to visualize Julia's fractals, with an interface such as :

Figure: WEB interface

WEB interface description :

1. A => Dynamic title with X and Y values
2. B => X axis: allows you to change the value of X
3. C => Y axis: allows you to change the Y value
4. D => Display option and original image download button
5. E => Fractal explorer, with high zoom capability.

# 2. General approach

There are 4 steps to follow:

## 1. Create a table of the number of iterations for each pixel in the image

- Tool: CUDA (C / C++)  
  
- OS: Linux (WSL 2 Ubuntu)  
  
- Hardware: Nvidia 8 GB RAM graphics card

## 2. Transformation of the iteration number array into images and compression of the array to optimize hard disk usage.

- Tools: CUDA (C / C++) and python 3  
  
- OS: Linux (WSL 2 Ubuntu)  
  
- Hardware: Nvidia 8 GB RAM graphics card

## 3. Creating a zoomable image with "openseadragon" and "deepzoom.py" software

- Tool: python 3  
  
- OS: Linux (WSL 2 Ubuntu) or Windows

## 4. Creation of a website to visualize fractals

- Tools: python 3 / HTML / JS  
  
- OS: Linux (WSL 2 Ubuntu) or Windows

# 3. Prerequisites

## 1. Activate WSL2 and NVIDIA (documentation)

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

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

## 2. Ubuntu commands

# Installing NVIDIA drivers and toolkits  
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  
  
# Pip3 installation  
sudo apt install python3-pip  
  
# Installing Python libraries  
pip3 install numpy numba pillow joblib py7zr

## 3 Estimating calculation parameters

We use the graphics card's 3D calculation function to tile the image as follows:

* tile size is a multiple of 256
* The number of tiles per side is defined as: floor(sqrt(tile size)) :
* The dimension of the image is therefore: (number of tiles per side) \* (tile size)
* This is the size of the uncompressed image:
* and also the size of the uncompressed binary in int (32 bits) :
* and also the size of the long uncompressed binary (64 bits):

As the graphics cards available have a maximum of 80 GB RAM :

* For INT calculations (32 bits) :
* For LONG (64-bit) calculations :

# 4. CUDA: creating a calculation program with several GPUs

The cuda code enables NVDIA GPUs to be used as computing centers.

The code I propose is broken down into 5 parts:

## 1. The header

This is the common code between cuda and c++, and includes :

* The type of fractal to generate: Type\_Fractal

// Definition of enumeration for fractal type  
enum Type\_Fractal { Mandelbrot, Julia };

* The **Complex** structure for representing complex numbers

// Definition of the Complex structure to represent complex numbers  
struct Complex  
{  
 double x, y; // Real and imaginary parts  
  
 // Constructor for initializing a complex number  
 \_\_host\_\_ \_\_device\_\_  
 Complex(double a = 0.0, double b = 0.0) : x(a), y(b) {}  
  
 // Overload operator + for addition of two complex numbers  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator+(const Complex &other) const  
 {  
 return Complex(x + other.x, y + other.y);  
 }  
  
 // Overload operator - for subtraction of two complex numbers  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator-(const Complex &other) const  
 {  
 return Complex(x - other.x, y - other.y);  
 }  
  
 // Overload the \* operator to multiply two complex numbers  
 \_\_host\_\_ \_\_device\_\_  
 Complex operator\*(const Complex &other) const  
 {  
 return Complex(x \* other.x - y \* other.y, x \* other.y + y \* other.x);  
 }  
  
 // Function for calculating the norm of a complex number  
 \_\_host\_\_ \_\_device\_\_ double norm() const  
 {  
 return sqrt(x \* x + y \* y);  
 }  
  
 // Function to raise a complex number to a given power  
 \_\_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));  
 }  
};

* **ParameterPicture** structure for storing fractal image parameters

// Define ParameterPicture structure to store fractal image parameters  
struct ParameterPicture  
{  
 long lenG; // Global length in 3D  
 long lenL; // Local length in 2D  
 double2 start; // Image starting point  
 double size; // Size of one side of the image  
 Type\_Fractal type\_fractal; // Fractal type (Mandelbrot or Julia)  
 double2 coef\_julia; // Julia fractal coefficients  
 double power\_value; // Power value  
 long iter\_max; // Maximum number of iterations  
 long id; // Image identifier  
  
 // Constructor to initialize a ParameterPicture object  
 \_\_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) {};  
  
 // Function to obtain 3D image size  
 \_\_host\_\_ \_\_device\_\_ size\_t Get\_size\_array\_3D() const  
 {  
 return (size\_t)lenG \* (size\_t)lenG \* (size\_t)lenG;  
 }  
  
 // Function to obtain 2D image size  
 \_\_host\_\_ \_\_device\_\_ size\_t Get\_size\_array\_2D() const  
 {  
 return (size\_t)lenG \* (size\_t)lenG \* (size\_t)lenL \* (size\_t)lenL;  
 }  
  
 // Function to obtain the position in double coordinates in the 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);  
 }  
  
 // Function to obtain the position in long coordinates in the 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);  
 }  
  
 // Function to obtain the 3D index of a position in the 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;  
 }  
  
 // Function to obtain the 2D index of a position in the 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;  
 }  
  
 // Function to set a value in the image data at a given position  
 \_\_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;  
 }  
 }  
  
 // Function to obtain an image data value at a given position  
 \_\_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;  
 }  
 }  
  
 // Function to print image parameters to a file  
 \_\_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. The cuda code

This is the code that calculates the Julia or Mandelbrot fractal:

* Kernel\_Picture: CUDA kernel for generating a fractal image

// CUDA kernel to generate a fractal image  
\_\_global\_\_ void Kernel\_Picture(ParameterPicture parameter\_picture, long \*data)  
{  
 // 3D index calculation for each 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;  
  
 // Get the corresponding 2D index  
 long index = parameter\_picture.Get\_index\_2D(idx, idy, idz);  
  
 // If index is valid  
 if (index >= 0)  
 {  
 // Get the corresponding complex position  
 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);  
  
 // If fractal type is Julia, use Julia coefficients  
 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;  
  
 // Calculate the number of iterations for the fractal  
 while (z.norm() < 2.0 && iter < parameter\_picture.iter\_max)  
 {  
 z = z.power(parameter\_picture.power\_value) + c;  
 iter++;  
 }  
  
 // Store the number of iterations in the data array  
 data[index] = iter;  
 }  
}

* RUN: the function to run the CUDA kernel

// Function to run the CUDA kernel  
cudaError\_t RUN(ParameterPicture parameter\_picture, long \*datas, int id\_cuda)  
{  
 // Calculate data size to be allocated  
 size\_t size = parameter\_picture.Get\_size\_array\_2D() \* sizeof(long);  
 long \*dev\_datas = 0;  
 cudaError\_t cudaStatus;  
  
 // Define thread and block configuration  
 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);  
  
 // Select GPU to use  
 cudaStatus = cudaSetDevice(id\_cuda);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  
 goto Error;  
 }  
  
 // Allocate memory on GPU for data  
 cudaStatus = cudaMalloc((void \*\*)&dev\_datas, size);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaMalloc failed!");  
 goto Error;  
 }  
  
 // Launch CUDA kernel  
 Kernel\_Picture<<numBlocks, threadsPerBlock>>(parameter\_picture, dev\_datas);  
  
 // Check if kernel launch has failed  
 cudaStatus = cudaGetLastError();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "Kernel\_Picture launch failed: %s\n", cudaGetErrorString(cudaStatus));  
 goto Error;  
 }  
  
 // Wait for kernel execution to complete  
 cudaStatus = cudaDeviceSynchronize();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching Kernel\_Picture!\n", cudaStatus);  
 goto Error;  
 }  
  
 // Copy data from GPU to host memory  
 cudaStatus = cudaMemcpy(datas, dev\_datas, size, cudaMemcpyDeviceToHost);  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaMemcpy failed!");  
 goto Error;  
 }  
  
 // Free memory allocated on the GPU  
 cudaFree(dev\_datas);  
  
 // Reset the GPU  
 cudaStatus = cudaDeviceReset();  
 if (cudaStatus != cudaSuccess)  
 {  
 fprintf(stderr, "cudaDeviceReset failed!");  
 return cudaStatus;  
 }  
  
 return cudaSuccess;  
  
Error:  
 // In the event of an error, free the memory allocated on the GPU  
 cudaFree(dev\_datas);  
 return cudaStatus;  
}

## 3. C++ code

This is the code used to manage the creation of Julia or Mandelbrot fractals:

* File\_Generate: the structure for managing files (.bin and .txt)

// Structure for generating files  
struct File\_Generate  
{  
 std::string bin, txt; // Binary and text file paths  
 bool exist; // Indicator whether file exists  
 File\_Generate(std::string bin, std::string txt) : bin(bin), txt(txt) {}  
};

* RUN: External CUDA function declaration

// External CUDA function declaration  
extern cudaError\_t RUN(ParameterPicture parameter\_picture, long \*datas, int id\_cuda);

* CreateFolder: Function to create the working folder

// Function to create the working folder  
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 gives rwx access rights for all  
 std::cout << "Directory created: " << dirName << std::endl;  
 }  
 else  
 {  
 std::cout << "Failed to create directory!" << std::endl;  
 }  
  
 return dirName;  
}

* if\_file\_exist: Function for checking whether a file exists

// Function to check if a file exists  
bool if\_file\_exist(const std::string &name)  
{  
 std::ifstream f(name.c\_str());  
 return f.good();  
}

* write\_bin: Function for writing binary data to a file

// Function for writing binary data to a file  
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: Supervision function for launching fractal calculations

// Supervision function for launching fractal calculations   
File\_Generate run(ParameterPicture parameter\_picture, std::string baseDir, int id\_cuda)  
{  
 // Create file paths  
 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";  
  
 // Initialize File\_Generate structure  
 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: Function for obtaining the number of existing binary files

// Function to obtain the number of existing binary files  
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: Function to open a text file and read its contents

/ Function to open a text file and read its contents  
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 << "contains" << myText << std::endl;  
 }  
  
 MyReadFile.close();  
 return out;  
}

* Main: Main function executed at launch

int main()  
{  
 //int(sqrt(lenG)) is the number of tiles per side.  
 //example for 720 ==> there are int(sqrt(720)) = 26 tiles so 26\*720 = 18,720 px per side, i.e. a total image size of 350,438,400 px  
 //therefore a binary file of 2,803,507,200 octes or 2.8 GB.  
 const long lenG = 720;  
  
 // max number of binary files not processed by the python script  
 const int max\_bin\_files = 4;  
  
 //Borne min max of X  
 const double coef\_x\_min = -1.5;  
 const double coef\_x\_max = 1.5;  
  
 //No iteration of X and Y  
 const double coef\_pas = 0.1;  
  
 // Check existence of id\_cuda.txt file  
 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 exist " << path\_file\_id\_cuda << std::endl;  
 return 1;  
 }  
  
 // Check existence of min.txt file  
 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;  
 }  
  
 // Check that max.txt file exists  
 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 of the base directory name  
 std::string baseDir = "datas\_" + id\_cuda\_str + "\_" + std::to\_string(lenG) + "p";  
 long id = 0;  
  
 // Loops to generate files for different coef\_x and coef\_y values  
 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\_step)  
 {  
 std::cout << "id = " << id << std::endl;  
 std::cout << "Get\_nbfiles\_bin " << Get\_nbfiles\_bin(Files\_G) << std::endl;  
  
 // Waits if the number of existing binary files exceeds the limit  
 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. The script to compile the program.

This is the script used to generate the 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. Parameters

These are the program's external calculation parameters:

* The id of the nvidia card to use from 0 to N, where n is the number -1 of available graphics cards
* The minimum bound of Julia's coef y
* The upper bound of Julia's coef y

## 6. Running the program

Compilation of code from directory **01-02\_Creation\_Datas\_et\_Images\_full** :

$ bash ./make\_main.sh

Code execution from directory **01-02\_Creation\_Datas\_et\_Images\_full** :

$ ./main

# 5. PYTHON 1: Creating life-size images with multiple GPUs

## 1. Booksellers :

# Fast table management library  
import numpy   
# Library for managing the creation of cuda functions directly in python  
from numba import jit, cuda  
# Image creation and manipulation library  
from PIL import Image  
# Stopwatch management library  
import time  
# Thread creation library  
from joblib import Parallel, delayed  
# Basic libraries  
import os  
from math import \*  
import gc  
# 7zip file creation libraries  
import py7zr

## 2. Constants :

# Number of Threads launched  
NB\_THREAD = 1  
#File names constant  
FILE\_TXT = "parameters.txt  
FILE\_BIN = "data.bin  
FILE\_ZIP = "data.zip  
FILE\_7ZIP = "data.7z"  
FILE\_NB\_TIF= "out1.tif"  
FILE\_G\_TIF= "out2.tif"

## 3. The ParameterPicture class and its reader :

class ParameterPicture :  
 id=0  
 lenG=0  
 lenL=0  
 start\_x=0  
 start\_y=0  
 size=0  
 type\_fractal=0  
 coef\_julia\_x=0  
 coef\_julia\_y=0  
 power\_value=0  
 iter\_max=0  
  
def generate\_ParameterPicture(path\_file\_txt:str):  
 f = open(path\_file\_txt, "r")  
 param = ParameterPicture()  
 while True:  
 line = f.readline()  
 if line == "":  
 break  
 exec("param. "+line)  
 f.close()  
 return param

## 4. cuda functions :

# Transforms an array of iterations into a two-color image (N & B)   
@jit(nopython=True)  
def lineariser\_2\_cuda(input\_array):  
 # Initialize output array with same shape as input  
 output\_array = numpy.empty\_like(input\_array, dtype=numpy.uint8)  
   
 # Browse input array  
 for i in range(input\_array.shape[0]):  
 for j in range(input\_array.shape[1]):  
 # Apply transformation out = in % 2 \* 255  
 output\_array[i, j] = (input\_array[i, j] % 2) \* 255  
   
 return output\_array  
  
# Transforms an array of iterations into a grayscale image [0 -> 255].  
@jit(nopython=True)  
def lineariser\_255\_cuda(input\_array, iter\_max:int):  
 # Initialize output array with same shape as input  
 output\_array = numpy.empty\_like(input\_array, dtype=numpy.uint8)  
  
 coef = ceil(iter\_max/255)   
  
 # Browse input array  
 for i in range(input\_array.shape[0]):  
 for j in range(input\_array.shape[1]):  
 # Retrieve case value  
 value = (input\_array[i, j])  
 # Make sure value is within range  
 value = max(0, min(iter\_max, value))  
 # Normalize the value in the range [0, 1].  
 normalized = (value) / (iter\_max)  
 # Add coef  
 phase = normalized \* coef  
 # Apply transformation   
 output\_array[i, j] = int(phase\*255) %255  
   
 return output\_array

## 5. python functions :

* save\_tif: Create a tif image on hard disk.

def save\_tif(image, filename):  
 # Save image as TIFF with compression  
 image.save(filename, format='TIFF', compression='tiff\_lzw')

* lister\_paths\_bin: lists binary files in a folder.

def lister\_paths\_bin(baseDir:str,nb\_thread:int,no\_thread:int):  
 list\_paths= list()  
 for path, subdirs, files in os.walk(baseDir):  
 for name in files:  
 if name==FILE\_BIN:  
 num = int(path.split("id\_")[-1])  
 if num % nb\_thread == no\_thread:  
 liste\_paths.append(path)  
 return liste\_paths

* bin\_file2image\_np\_2D: transforms a binary file into a 2D array.

def bin\_file2image\_np\_2D(file\_bin:str,param:ParameterPicture):  
 data = numpy.zeros(0)  
 start\_l = time.time()  
 with open(file\_bin, 'rb') as f:  
 data = numpy.fromfile(f, dtype=numpy.uint64)  
 elapsed = time.time() - start\_l  
 print(f'Time d'\'execution Open : {elapsed:.4} s')  
  
 start\_l = time.time()  
 data = data.reshape(param.lenG\*param.lenL,param.lenG\*param.lenL)  
 elapsed = time.time() - start\_l  
 print(f'Execution time 1d --> 2D : {elapsed:.4} s')  
 return data

## 6. main functions :

def sub\_main\_bin\_2\_tif(value:int):  
 nb\_thread = NB\_THREAD  
 while True:  
 if True:  
 baseDir = "./"# makeBaseDir(nbpts)  
 liste\_paths = lister\_paths\_bin(baseDir,nb\_thread,value)  
 for path\_bin in liste\_paths:  
 file\_txt = os.path.join(path\_bin, FILE\_TXT)  
 file\_bin = os.path.join(path\_bin, FILE\_BIN)  
 file\_7zip = os.path.join(path\_bin, FILE\_7ZIP)  
 file\_nb\_tif = os.path.join(path\_bin, FILE\_NB\_TIF)  
 file\_g\_tif = os.path.join(path\_bin, FILE\_G\_TIF)  
   
 if os.path.isfile(file\_txt) and os.path.isfile(file\_bin):  
 start\_l = time.time()  
 ########################### generate\_ParameterPicture ######################  
 start = time.time()  
 print(f "Use : {file\_txt}")  
 param = generate\_ParameterPicture(file\_txt)  
 end = time.time()  
 elapsed = end - start  
 print(f'Execution time generate\_ParameterPicture: {elapsed:.4} s')  
   
 ########################### bin\_file2image\_np\_2D ######################  
 start = time.time()  
 image\_raw = bin\_file2image\_np\_2D(file\_bin,param)  
 end = time.time()  
 elapsed = end - start  
 print(f'Execution time bin\_file2image\_np: {elapsed:.4} s')  
 max = numpy.max(numpy.max(image\_raw))  
 gc.collect()  
  
 ########################### create img B&W % 2 ######################  
 print("create img B&W % 2 ")  
  
 start = time.time()  
 image\_np\_lineariser = lineariser\_2\_cuda(image\_raw)  
 end = time.time()  
 elapsed = end - start  
 print(f'Execution time lineariser\_2\_cuda: {elapsed:.4} s')  
  
 im\_out = Image.fromarray(image\_np\_lineariser.astype('uint8')).convert('L')  
 save\_tif(im\_out,file\_nb\_tif)  
   
 del image\_np\_lineariser  
 del im\_out  
 gc.collect()  
  
 ########################### create img B&W % 255 ######################  
 print("create img B&W % 255 ")  
  
 start = time.time()  
 image\_np\_lineariser = lineariser\_255\_cuda(image\_raw,max)  
 end = time.time()  
 elapsed = end - start  
 print(f'Execution time lineariser\_255\_cuda: {elapsed:.4} s')  
   
  
 im\_out = Image.fromarray(image\_np\_lineariser.astype('uint8')).convert('L')  
 save\_tif(im\_out,file\_g\_tif)  
  
 del image\_np\_lineariser  
 del im\_out  
 del image\_raw  
 gc.collect()  
  
 ########################### create 7zip datas ######################  
  
 print("Create a 7ZipFile Object")  
 start = time.time()  
 with py7zr.SevenZipFile(file\_7zip, 'w') as z:  
 z.writeall(file\_bin)  
 elapsed = time.time() - start  
 print(f'7zip execution time : {elapsed:.5} s')  
   
 # Check to see if the zip file is created  
 if os.path.exists(file\_7zip):  
 print("7ZIP file created")  
 os.remove(file\_bin)  
 else:  
 print("7ZIP file not created")  
 elapsed\_l = time.time() - start\_l  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print(f'Execution time {path\_bin}: {elapsed\_l:.5} s')  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

## 6. main function dispatcher :

def main\_bin\_2\_tif():  
 f = open("parameters/id\_cuda.txt", "r")  
 id\_cuda = int(f.readline())  
 f.close()  
 print(f" id cuda = {id\_cuda}")  
 cuda.select\_device(id\_cuda)  
  
 nb\_thread = NB\_THREAD  
 values = range(0,nb\_thread)  
 Parallel(n\_jobs=nb\_thread,prefer="threads")(delayed(sub\_main\_bin\_2\_tif)(value) for value in values)

## 7. script launch :

if \_\_name\_\_ == "\_\_main\_\_":  
 start\_g = time.time()  
 main\_bin\_2\_tif()  
 end\_g = time.time()  
 elapsed = end\_g - start\_g  
 print(f 'Execution time G: {elapsed} s')

# 6. PYTHON 2: DZI creation

We use [deepzoom3](https://github.com/muranamihdk/deepzoom3), which has been updated to take account of changes in the PIllow library.

[deepzoom3](https://github.com/muranamihdk/deepzoom3) transforms a large image into small tiles (1024 px) for optimized use on a website. a bit like google map.

the web engine to be used is [openseadragon](https://openseadragon.github.io/)

the code that performs this task is **Sub\_03\_Export\_Web.py** with the **lib\_deepzoom.py** library.

# 7. WEB : WEB site creation

the code that performs this task is **Sub\_04\_Index\_DZI.py**

It creates a *.js* file containing a list of *.dzi* files and full-size images.

Axis/range limits are calculated automatically, and all that's left is to manually enter the step in the **index.html** file.

<!-- X axis -->  
<input type="range" class="custom-range" id="range\_x" step="50" onchange="update\_plot()"  
value="10">  
  
<!-- Y axis -->  
<input type="range" class="custom-range" id="range\_y" step="50" onchange="update\_plot()"  
value="10">

To launch the local web server, run the script **run\_web.sh** in the folder **./03-04\_Export\_WEB\_et\_site\_web/Web** and open the url [http://localhost:8000/.](http://localhost:8000/)