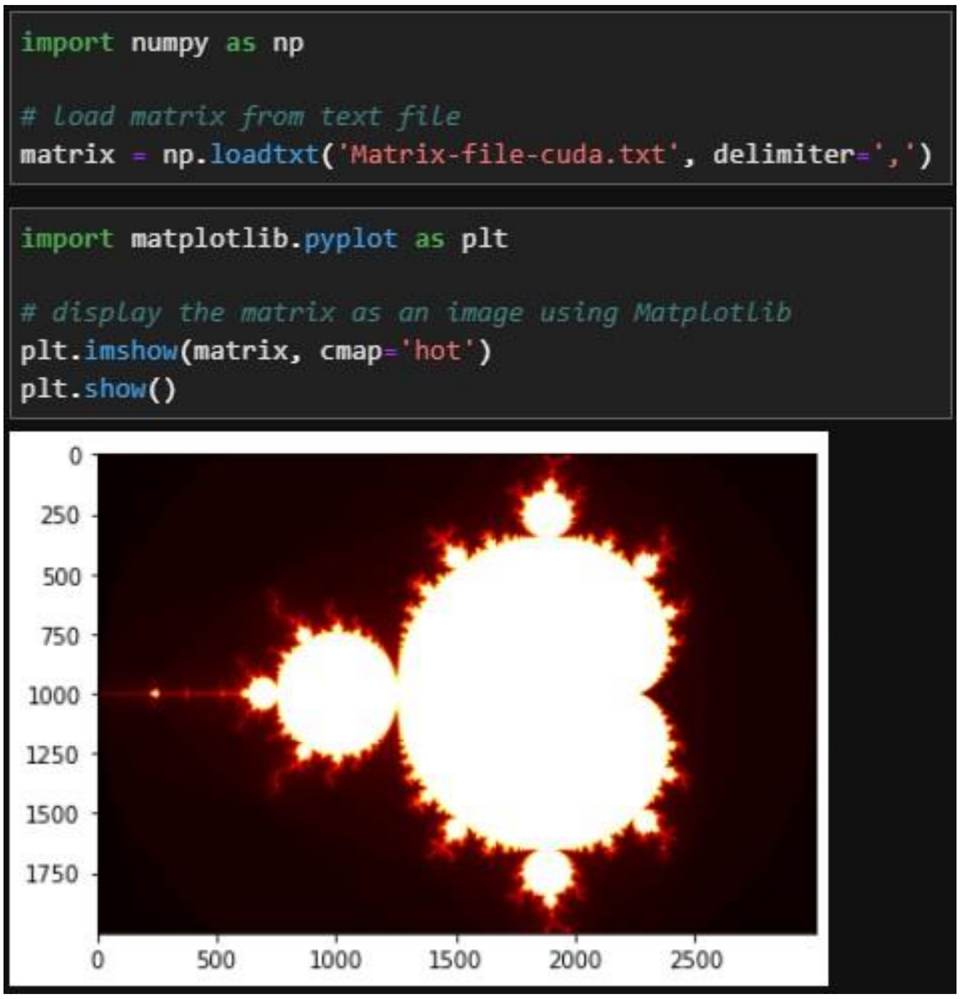
Authors: Manuel Parmiggiani & Giacomo Garbarino

High Performance Computing Final Project: Mandelbrot Set

This paper will discuss and compare different implementations of the algorithm for generating the Mandelbrot set in order to state which methodology suits better this kind of problem, the behavior of each implementation can be sequential or parallel depending of the employed approach.
Each of the presented approaches has its own implementation with the following common features:

- The parameters used for generating the Mandelbrot set (such as image ratio, image size, ranges of the set, ecc...) are defined as constants;
- The program computes the elapsed time between the start and the end of the algorithm, including the time for allocating the required data structures.
- After computing the Mandelbrot set for the chosen constant parameters, the program will produce a txt file containing the generated set, this file can then be given as an input to a python script that visualizes it as a 2D image corresponding to the set:



```
import numpy as np

# Load matrix from text file
matrix = np.loadtxt('Matrix-file-cuda.txt', delimiter=',')
```

```
import matplotlib.pyplot as plt

# display the matrix as an image using Matplotlib
plt.imshow(matrix, cmap='hot')
plt.show()
```

Sequential Implementation

The first and the simplest (but not the better) methodology for implementing the algorithm, int this implementation the generated image is scanned and updated sequentially. To compile and run the program, run the following commands over the CUDAcluster:

- `nvcc -std=c++11 -O0 Sequential-Code.cpp -o sequential`
- `./sequential`

The program takes no injected parameters since all of the constants were already presentin the source code, also the O0 flag disables the automatic optimization made by the compiler in order to observe the execution time of a pure serial execution. The results of this first execution will be then compared with the execution times of an optimized executable of the same source code, this can be done by replacing the O0 flag with another one which enables vectorization:

- `nvcc -std=c++11 -O2 Sequential-Code.cpp -o sequential_vectorized`
- `./sequential_vectorized`

The compiler offers a wide range of optimizations (usually from O1 to O5) but in this case the choice is even because all of the optimized versions return the same execution time. The advantage of employing an optimized version will be measured in terms of speedup, computed by dividing the execution time of the sequential execution for the execution time of the optimized execution. To also discuss the scalability of the presented solutions, the code was compiled and run multiple times by changing the “RESOLUTION” constant every time, a bigger value of this constant implies a larger width and height of the image and so a bigger problem size. The following table presents and compares the obtained results:

Resolution	Non-Optimized (O0)	Vectorized (O2)	Speedup
500	8416 ms	6433 ms	1.31
1000	33556 ms	25350 ms	1.32
5000	842501 ms	633453 ms	**1.33**
10000	3366804 ms	2555512 ms	1.32

From the results above, the following conclusions can be inferred:

- The vectorized version is (obviously) faster than the non-optimized one.
- The gained optimization isn't proportional with the problem size since, even with a significant difference in the image dimensions, the obtained speedup is almost the same.

The computational time used for comparing the sequential implementation with the following one will be the Vectorized time since it's the better that can be done for the sequential scenario.

CUDA Implementation

Compute Unified Device Architecture (CUDA) provides the possibility of employing a GPU to accelerate even more the computation. The first thing to do to translate the sequential code in a correct CUDA code is to find the portion of code being the hot-spot of the program, in this case the hot-spot is identified in the nested for instructions executing the algorithm, those nested iterations have to be moved into a CUDA kernel function in order to parallelize the execution. After that, there is the need to settle an appropriate data structure for the image, this can be done through the *cudaMallocManaged()* function, this function is preferable to the classic *cudaMalloc()* since it provides unified memory, making the image matrix accessible both from the CPU and the GPU without having to transfer it with functions like *cudaMemcpy()*; this reduces the computational overhead of transferring the data from the CPU to the GPU and lowers the execution time even more. Last thing, the algorithm moved to the kernel function had to be slightly modified to adapt to the CUDA execution environment; also, the <complex> datatype couldn't be used since the compiler discouraged its adoption inside a kernel function, so each complex variable was replaced with a couple of new variables representing the real and the imaginary part of that complex number. To compile and run the program, run the following commands over the CUDA cluster:

- nvcc -std=c++11 CUDA-Code.cu -o cuda
- ./cuda *threads*

The program takes as an injected parameter the number of threads, there is a specific rule to follow to decide how many threads run: the GPU multiprocessor schedules threads in groups of 32 threads so the number of threads has to be a multiple of 32 and cannot exceed 1024 which is the maximum number of runnable threads. The following table presents the results obtained from the execution of the CUDA implementation, each column of the table corresponds to the total number of threads and each row to the resolution of the image, each cell of the table contains the speedup with respect to the time saved from the sequential execution against an image of the same dimension for that number of threads:

Image Resolution	32 Threads	64 Threads	128 Threads	256 Threads	512 Threads	1024 Threads
500	12.28	12.92	**13.51**	13.49	13.46	13.40
1000	52.81	52.92	**53.26**	53.03	53.14	52.7
5000	1177.42	1308.79	1322.45	1325.22	**1327.99**	1322.45
10000	4776.66	5030.54	5204.71	**5236.70**	5215.33	5194.13

The huge speedup obtained highlights that CUDA perfectly fits the needs of this type of problem, it is possible to observe how the optimal number of threads scales up with the problem size, ending every time with almost the same computational time andleading to a gradually better speedup.