TP01
Introduction to parallel processing

12/14/21

Abstract

In this practical work, we will see how to convert a simple iterative process into a parallel process.

1 Introduction

First, generate the Visual Studio solution, compile it and run the TP01 program. You'll find the source of the practical work by cloning this git repository:

```
https://github.com/robinfaurypro/GPGPU_ISIMA_2021-2022.git
```

The CMakeLists file is stored into the TP01 folder. Open a terminal on your working folder and use this cmake command:

```
cmake -G "Visual Studio 15 2017 Win64" ..
```

If everything is going well, you can compile and run the TP01 executable. The application generate a grayscale image in 512 by 512 pixels.

2 Image generation

The image_data object store the interleaved data of the image. That mean the first value of the vector corresponding to the red value of a pixel, the second to the green and the third to the blue. Create a function to fill the image according to the picture. The x coordinate is assign to the red channel and the y coordinate to the green. The blue channel will be always zero. You can start with an image 256x256 if you prefer.

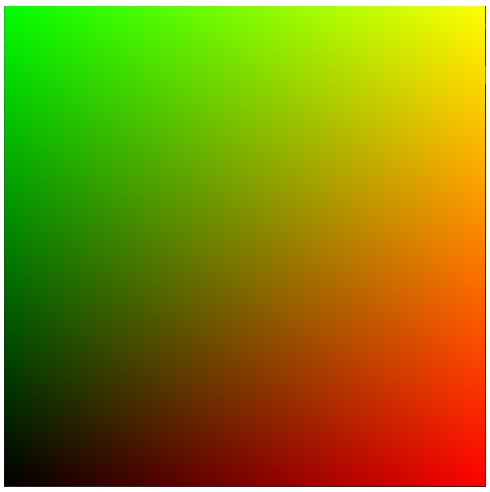

Figure 1: UV of the image

A GPU is efficient with float operation so it's a good idea to works with floating data. You can duplicate the input data into a vector of float and convert the $0 < x < 512$ into $0 < u < 1$ and $0 < y < 512$ into $0 < v < 1$.

Colors can also be saved as value between zero and one. You just have to multiply them by 256 at the very end of the process.
Create an image with a disc on it having the radius of half the image.

3 Critical path

In GPGPU, the most important part is to identify the critical path. Find it and create a function ”compute” to isolate this path. This function is our kernel.
We saw a GPU process kernel into warps. For this practical we will use std::thread as warp. Take a look at your configuration to know how many logical processors do you have. Create one std::thread per logical processors. A std::thread need a function as input. Create a ”dispatch” function that run the kernel for a subset of the image.

```
std::thread t0(
        dispatch,
        startX,
        endX,
        startY,
        endY,
        width,
        height,
        iterationMax);
//...
t0.join();
//...
```

At this step you need to set your buffer as a global variable if you want it to be shared between threads.

4 profiling

We should now profile the algorithm. For that you need to include chrono

```
#include <chrono>
```

and compute the duration of the process.

```
auto start = std::chrono::system_clock::now();
//...
auto end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end-start;
printf("duration: %f\n", elapsed_seconds.count());
```

Compare the performance between one thread and the maximum of logical thread. Does the algorithm scale linearly? What happens if you runing the algorithm on an non power of two resolution?

5 The Mandelbrot set

Let’s try to have a more heavy parallel algorithm. The Mandelbrot set is the set of complex numbers c for which the function

$$f_c(z) = z^2 + c \quad (1)$$

does not diverge when iterated from $z = 0 + 0i$.
As a reminder complex number multiplication is:

$$z_1 * z_2 = (z_1.real * z_2.real - z_1.imag * z_2.imag) + (z_1.real * z_2.imag + z_1.imag * z_2.real) * i \quad (2)$$

And the addition is:

$$z_1 + z_2 = (z_1.real + z_2.real) + (z_1.imag + z_2.imag) * i \quad (3)$$

Create a class Complex and override operator ’+’ and ’*’. Add a member function to compute the modulus of the complex too (You can also use the complex object from the stl).

Choose a square size for the output image (for example 1024x1024) and create a buffer of float to store all RGB pixel. A pixel is three float from 0.f to 1.f.
Now the aim is to run the function (1) for each pixel. First we need to convert each coordinate pixel to the complex plan.

```
Complex c(
        static_cast<float>(x)/static_cast<float>(width),
        static_cast<float>(y)/static_cast<float>(height)
);
```

We can run the function (1) while the result don't diverge. We consider a complex number is outside of the [-2; 2] window is a divergent value. *iterationMax* is the number max of "jump".

```
unsigned int cmp = 0u;
while (z.modulus() < 2 && cmp <= iterationMax) {
        z = z*z + c;
        ++cmp;
}
```

If cmp reach iterationMax that means the complex number chosen is part of the Mandelbrot set. We can set the color of this pixel to black. in the other case we can set the color to red. We can also use a gradient for a better result.

```
const float red = static_cast<float>(cmp)/static_cast<float>(iterationMax);
```

For your information the Mandelbrot set have a better look if c is padded by $-1.5 - 0.5i$.

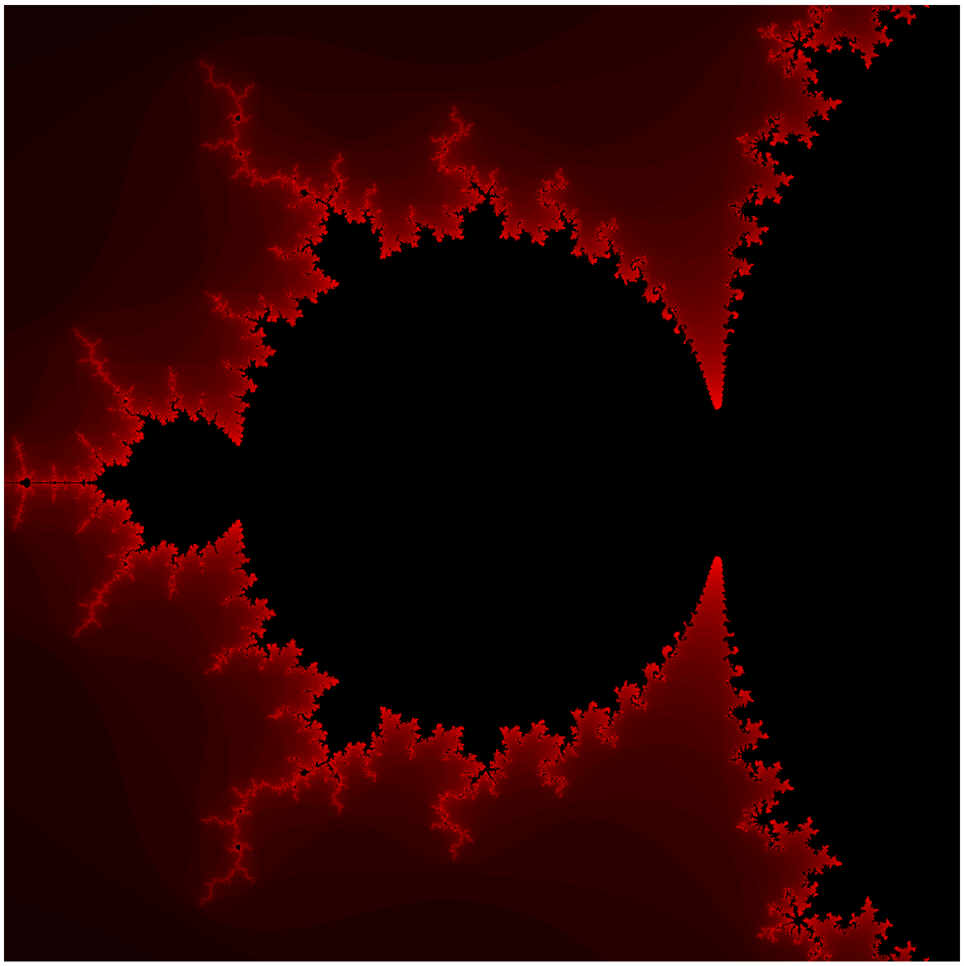

Figure 2: Mandelbrot set

6 Julia set

The Julia set is similar to the Mandelbrot set, but with a chaotic behaviour. That means iterationMax need to be increased if we want to compute the image. You can compute the fractal using those parameters:

- z = x/width + (y/height)*i
- c = 0.292 + 0.015*i
- iterationMax = 400

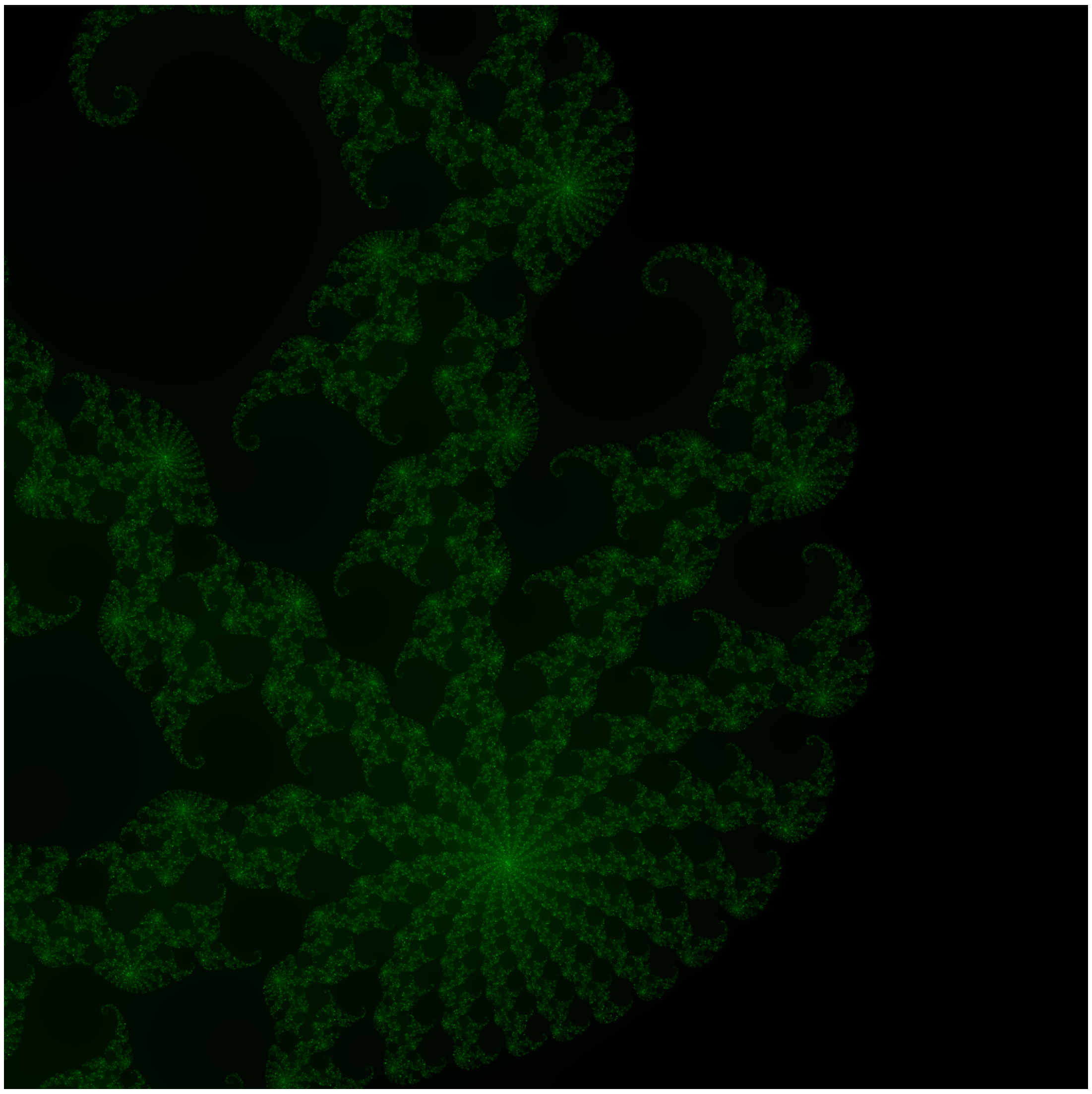

Figure 3: Julia set