UNIVERSITÀ DEGLI STUDI DI TRENTO

Advanced Computing Architectures

CUDA Accelerated Mandelbrot: Speeding up fractal generation with parallel GPU computation

Exam's project

Roberto Negro
[211505]

A.Y. 2020/2021

Contents

1 Introduction

The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2+c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value. Its definition is credited to Adrien Douady who named it in tribute to the mathematician Benoit Mandelbrot, a pioneer of fractal geometry.

Mandelbrot set images may be created by sampling the complex numbers and testing, for each sample point c, whether the sequence $f_c(0), f_c(f_c(0)), ...$ goes to infinity.

Treating the real and imaginary parts of c as image coordinates on the complex plane, pixels may then be coloured according to how soon the sequence $|f_c(0)|, |f_c(f_c(0))|, ...$ crosses an arbitrarily chosen threshold.[6]

The goal of this project is to improve the speed of the generation of the fractal by exploiting parallelization and fast float computations offered by GPUs. Due to the extreme speed up, the algorithm will be also enriched by several graphical improvements and the possibility to explore and modify the fractal in real time.

2 Basic algorithm

2.1 General setup

Since CUDA kernel code does not support extensively complex numbers and their specific operations, in the calculation involved in the generation of the fractal the complex numbers are treated by the real and imaginary part as two different double variables. To improve even more the performances, those computations are written without support functions, allowing the optimization offered by the compiler to easily improve further more the performances.

In order to exploit parallelization while generating the fractal, each point c, corresponding to a pixel in the image space, is flattened out and represented by an element in an 2-dimensional array (for simplified optimization): this array represent, in the basic version of the algorithm, if the corresponding point c is inside of the Mandelbrot set or not.

Each thread in CUDA has a specific index inside of the block of execution, and has access to the block index and dimensions. So, we can easily obtain the corresponding array index as $index = blockIndex * blockDimension + threadIndexInBlock$.

Then, the thread index is mapped to a specific point in the fractal space. This is

done with the following formula:

$$c_r = \left(\frac{(index\%width - 1) - \frac{width}{2}}{width * zoom} \right) * \frac{width}{height} + x$$

$$c_i = \frac{\frac{index}{width} - \frac{height}{2}}{height * zoom} + y$$

Where *width* and *height* are the image resolution, *zoom* is the zoom in the image (1 means no zoom, higher values increase the zoom, lower values reduce the zoom), x and y is the horizontal and vertical shift translations in the image (0 means no shift, negative values means left/bottom translation, positive values means right/top translation).

The code is developed and tested with an NVIDIA GTX 1070 Ti, which has 2432 CUDA cores, on Mac OS 10.13.6 (High Sierra, last compatible version of Mac OS with NVIDIA GPUs).

In general, images are generated two times bigger than the desired resolution, in order to introduce anti-aliasing via down-sampling.

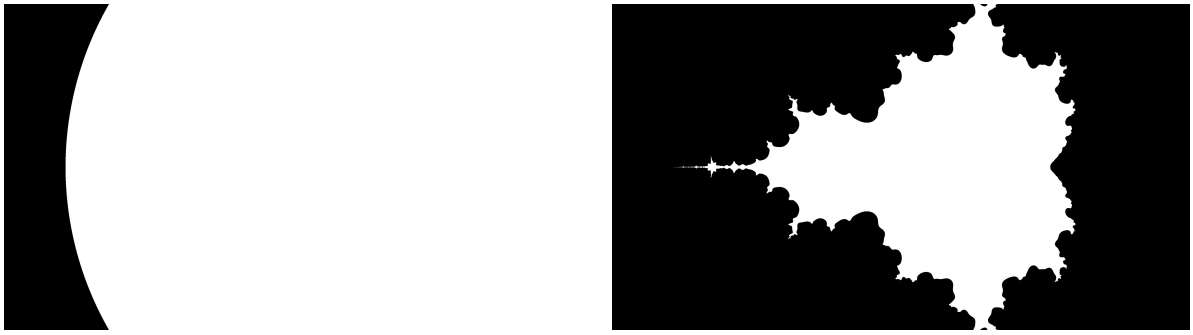
2.2 Fractal generation

Starting from the specific complex number representing the image point, the algorithm start, with $z = 0$, to iterate following the Mandelbrot set formula: $f_c(z) = z^2 + c$. A property of the Mandelbrot Set defines that a point c belongs to the Mandelbrot set if and only if $|z_n| \leq 2$ for all $n \geq 0$.[6] So, as soon as $|z_n|$ becomes higher than 2, we consider that point as outside of the Mandelbrot set and stop iterating. In order to allow the algorithm to be computable, we also need to define the maximum number of iterations after which we suppose that the point is inside of the Mandelbrot set.[8] Higher number of iterations are needed to produce more accurate approximation of the Mandelbrot set, as show in figure 2.1. This becomes useful in case of extreme zooms, as in figure 2.2. Obviously, the higher the number of maximum iterations, the higher the maximum time of execution needed in case of points that are inside of the Mandelbrot set, as shown in in figure 2.3.

2.3 Performances

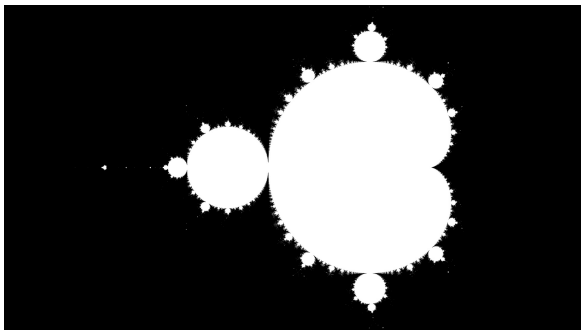
2.3.1 Aligned and Coalesced access

In order to increase as much as possible the performances of the algorithm, the algorithm was changed in order to have aligned and coalesced warp access. Currently, the image is treated as an R, G, B image with values from 0 to 255 for each channel. So, we can represent each pixel by $3 * sizeof(uint8) = 3$ bytes. Those three values are flattened out and are consecutive in the image array. So, thread 1 will set indexes $0, 1, 2$, while thread 2 will set indexes $3, 4, 5$, and so on. But doing so, the warp

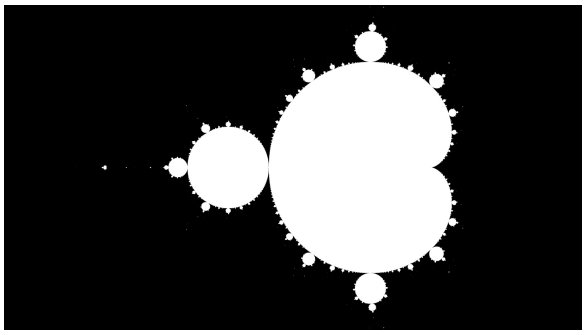


(a) 1 iteration

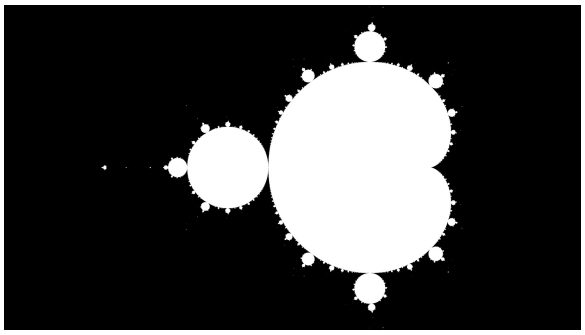
(b) 10 iterations



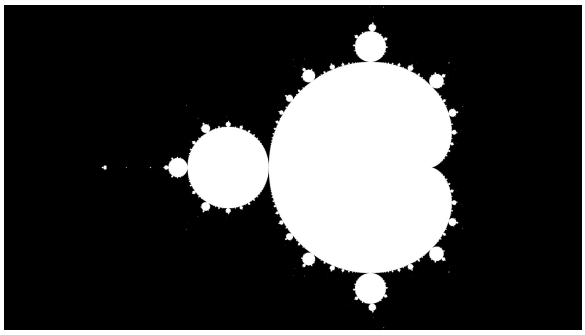
(c) 100 iterations



(d) 1000 iterations

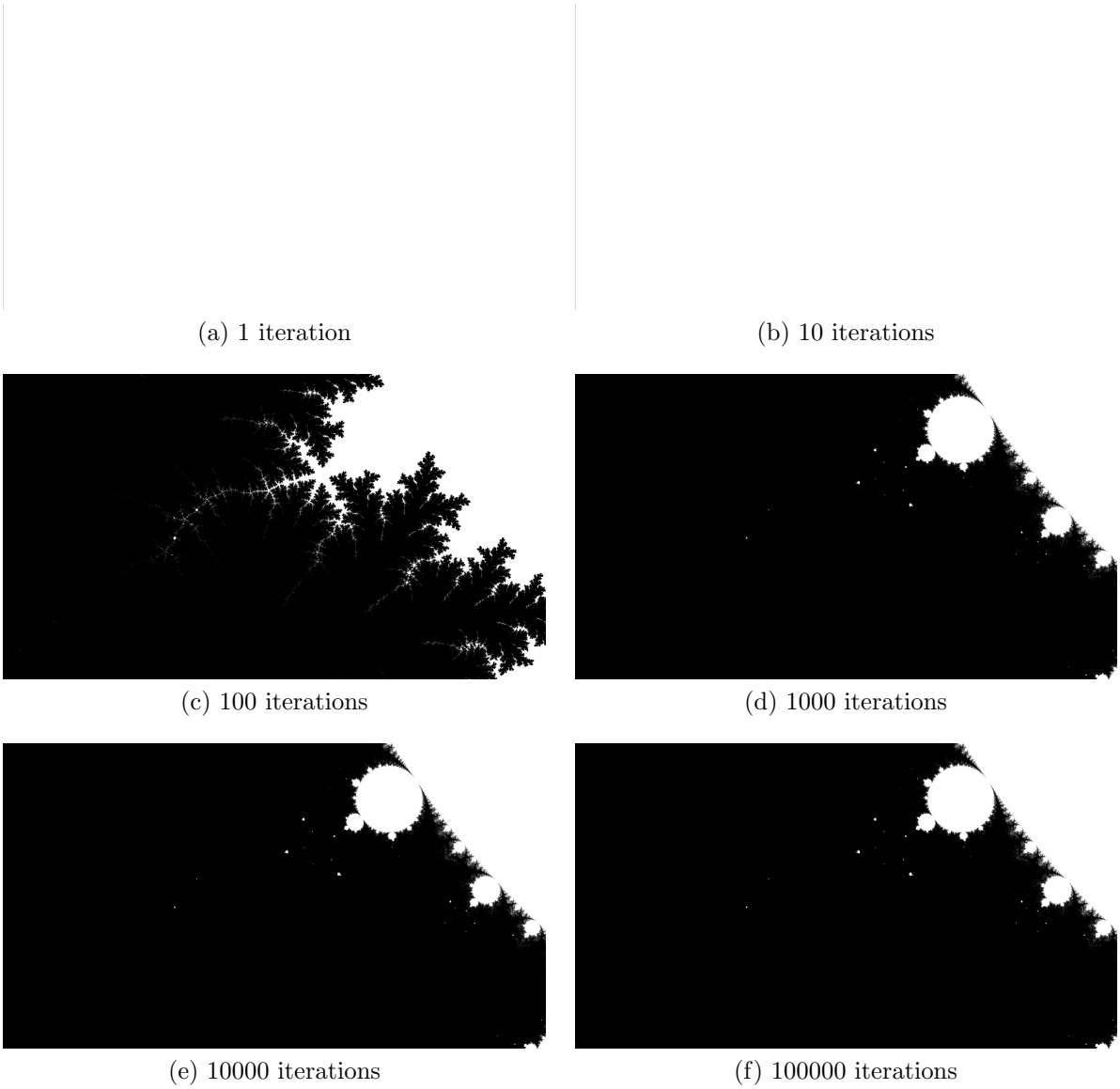


(e) 10000 iterations



(f) 100000 iterations

Figure 2.1: Mandelbrot (x: -0.6, y: 0.0, zoom: 0.5) with different number of maximum iterations



(a) 1 iteration

(b) 10 iterations

(c) 100 iterations

(d) 1000 iterations

(e) 10000 iterations

(f) 100000 iterations

Figure 2.2: Highly zoomed Mandelbrot (x: -1.36771667, y: 0.04114768, z: 47.26478546) with different number of maximum iterations

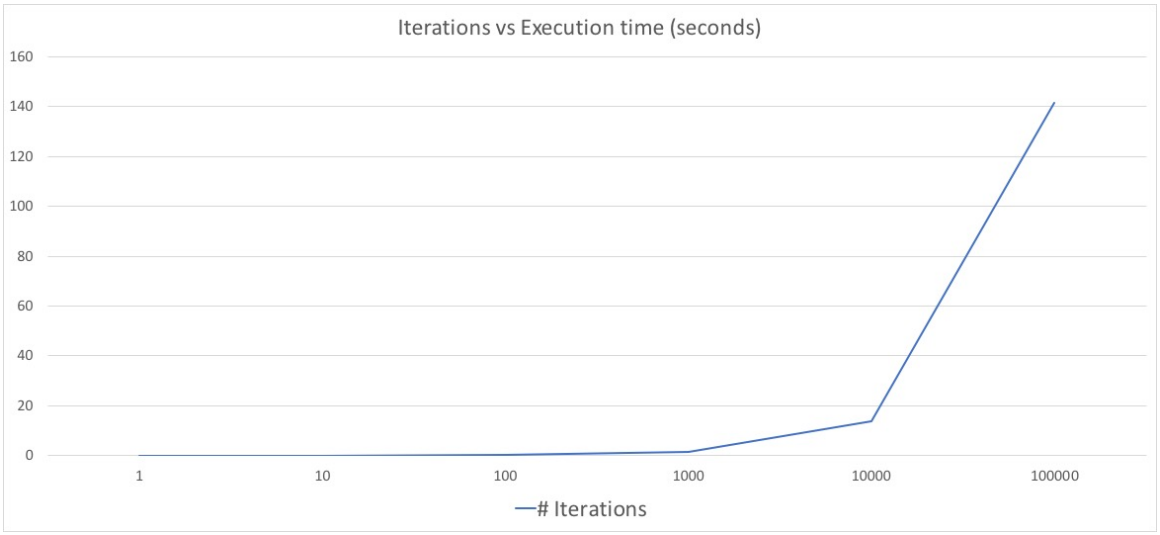


Figure 2.3: Iterations vs Execution time

access is not aligned: considering that the warp size is a multiple of 32 (in order to avoid wasted threads), the first warp will access the first 96 indexes, wasting the other $128 - 96 = 32$ indexes. But then the second warp has to access the memory two times: one to obtain the remaining 32 indexes (fetching indexes $[0, 127]$), and then the second to obtain the next 96 (fetching indexes $[128, 255]$). And then the problem propagates for all the other subsequent warps. To fix this, another $uint8$ was added after the three RGB colour channels. In this way, we represent each pixel by $4 * sizeof(uint8) = 4$ bytes, and this allows the warp access to be aligned and coalesced: thread 1 will access indexes $0, 1, 2, 3$, thread 2 will access indexes $4, 5, 6, 7$, ..., thread 32 will access indexes $124, 125, 126, 127$. This optimization, in this specific case, does not improve much the performances, because the algorithm is anyway pretty fast in terms of memory: there are no dependencies between neighbouring cells, and each colour value (of just 3 bytes) is written just once in the array at the end of the execution of the respective thread.

2.3.2 Unrolling

Another common technique to optimize device code is unrolling. In this particular case, due to the high variability in terms of time of execution of each thread inside of the same warp, unrolling does not help the performances. In particular, in situations where inside of the same warp there are threads that are covering only pixels that are all inside of the Mandelbrot set, those threads take a lot of time to finish. But since the threads of the same warp are synchronized, all the other threads have to wait for them to finish before proceeding on the next warp. So, by introducing unrolling, we increase further more the variability in terms of execution time of each thread: while the number of iterations possible for a thread without unrolling is $(1, MAX\_ITERAITONS]$, if we assign N pixels to a single thread (via unrolling), the thread will now range in $(1, N * MAX\_ITERATIONS]$ iterations.

Figure 2.4: Block size vs Execution time

2.3.3 Results

In order to find the best configuration between aligned+coalesced access VS not aligned access, unrolled VS not unrolled, and also to find the best block sizes, several tests were made. As a benchmark, a black and white Mandelbrot image of sizes 4000x2000 was generated, with those specific settings: Zoom 0.5, X -0.6, Y 0.0, number of maximum iterations: 100'000, bailout radius R set as 1000 (see section 3.1), and no approximation (see subsection 2.3.4). The results are summarized in the table 2.1.

So, by considering the overall best approach (in this scenario) of not using unrolling techniques, further analysis were done regarding the block size, as shown in figure 2.4. In the comparison, it's possible to see how the wasted memory access are more relevant with smaller block sizes, while with bigger block sizes the wasted memory required to align the accesses starts to affect negatively the performances. In conclusion, the best results were obtained with a warp size of 32, no unrolling, and aligned and coalesced wrap access. This configuration will be used for all the next analysis and generations.

2.3.4 Optimization: Approximation factor

In order to speed up the computations, we can introduce a small ϵ parameter that defines an approximation threshold. To do so, we add a second check in each iteration and, as soon as $|dz| < \epsilon$, we stop the iterations and consider the point as inside of the Mandelbrot Set (if the derivative is so small, it's improbable that $|z_n| > 2$).[1] Even if this adds some complexities due to the calculation of the derivative of z, it speeds up consistently the computation in the more critic interior areas (where, otherwise, for each pixel the maximum number of iterations are entirely executed), as shown in in figure 2.5. Obviously, the value of ϵ has to be regulated considering, more than the level of zoom required, the position in the space: points near the conjunction between

Grid Size	**Block Size**	**Unroll (pixels×thread)**	**Aligned + Coalesced**	**Execution time**
250000	32	1	Yes	14,10s
125000	64	1	Yes	14,19s
62500	128	1	Yes	14,37s
250000	32	1	No	14,41s
83334	96	1	Yes	14,43s
62500	128	1	No	14,51s
125000	64	1	No	14,66s
83334	96	1	No	14,78s
1953	128	32	No	34,55s
1953	128	32	Yes	35,06s
3906	64	32	Yes	39,68s
7812	32	32	No	40,64s
2604	96	32	Yes	40,91s
7812	32	32	Yes	41,09s
3906	64	32	No	41,53s
2604	96	32	No	43,02s
976	128	64	Yes	44,39s
3906	32	64	Yes	44,53s
1953	64	64	Yes	44,57s
1302	96	64	Yes	45,36s
976	128	64	No	45,58s
1953	64	64	No	45,64s
3906	32	64	No	45,89s
1302	96	64	No	46,74s
2604	32	96	Yes	55,86s
1302	64	96	Yes	56,18s
2604	32	96	No	56,53s
1302	64	96	No	57,17s
651	128	96	Yes	57,39s
868	96	96	No	58,06s
651	128	96	No	58,41s
868	96	96	Yes	58,55s

Table 2.1: Performance comparison

Figure 2.5: Approximation vs Execution time

the cardioid (the main "body") and the periodic bulbs are more affected by higher ϵ. Some examples can be seen in figure 2.6.

2.3.5 CPU vs GPU

The code was then changed in order to be also executed on CPU, in a multi-threaded fashion. The CPU used for the computations is an Intel i7 8700K, 6 cores 12 threads CPU. During the benchmarks the CPU was at a stable clock frequency of 4.30GHz (Turbo Boost). Obviously, by increasing too much the number of threads, we stop gaining performances pretty soon. Even if at a lower number of iterations a "real-time" ($< 1s$) rendering can be achieved also by the CPU, in complex highly zoomed scenes, where high number of iterations are needed, the GPU outperforms the CPU by a 15x factor (220 seconds for CPU vs 14 seconds for GPU, with 10000 iterations). The results are shown in figure 2.7.

3 Colouring

There are several different techniques that can be implemented in order to add colors and gradients to the Mandelbrot set. The easiest techniques are escape-time-based techniques, which assign a color to the outside points depending on the number of iterations needed to the orbit to escape (so we look at n such as $|z_n| > 2$).[1] Those techniques suffer from visible strands instead of smooth gradients, so they've been replaced by more advanced techniques.

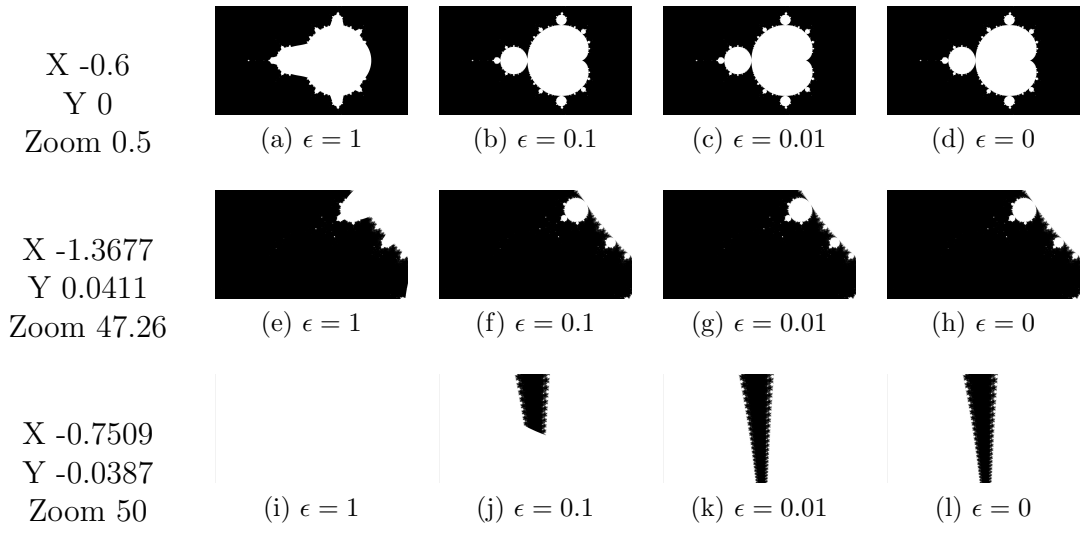


Figure 2.6: Different approximation levels. The third set of images are positioned right between the main cardioid and the 2-period bulb, the most sensitive area w.r.t. approximation.

3.1 Potential

By the introduction of the potential function, we can associates a real number V with each value of z. The potential function is defined, in this case, as:

$$V(c) = \lim_{n\to\infty} \left(\frac{\log_+ |f_c^n(z_0)|}{2^n} \right).$$

If we choose a large enough bailout radius R (e.g., 1000), we have that:

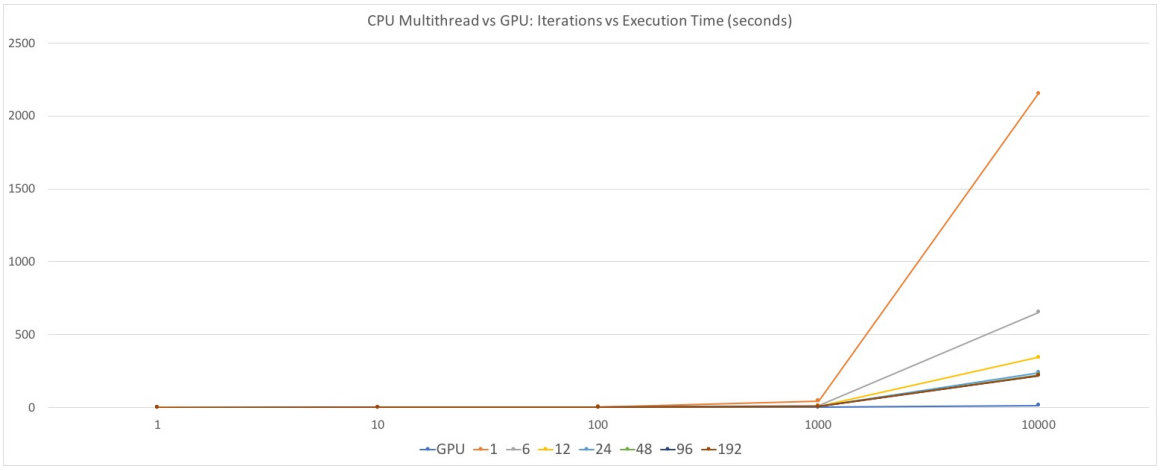
$$V(c) \approx \frac{\log(R)}{2^n}.$$

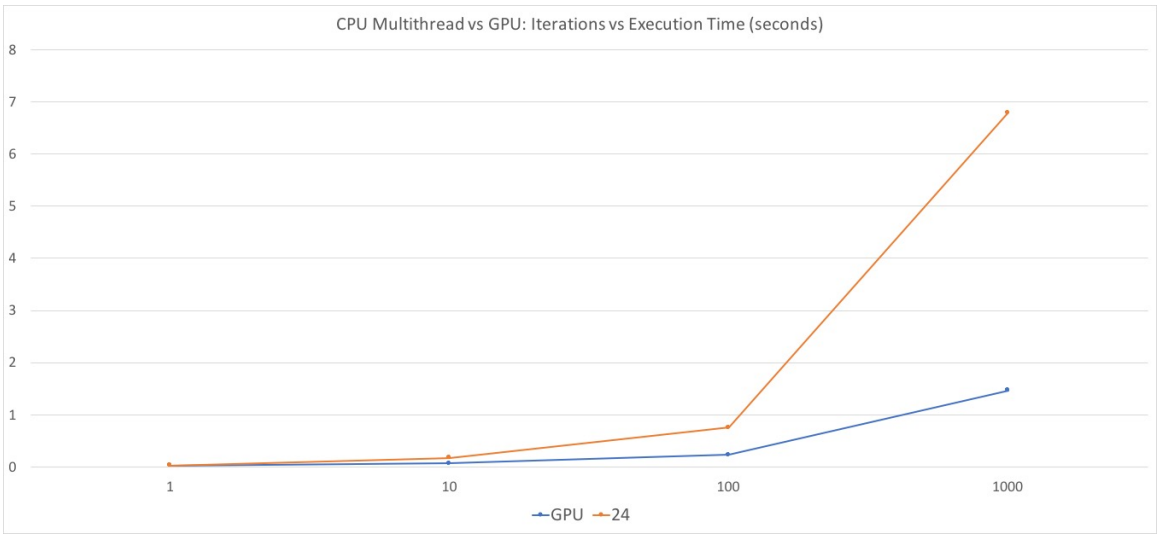
So, as soon as $|z| > R$, we also have a good approximation $V(c) \approx \frac{\log|z|}{2^n}$, and we can thus choose a continuous coloring scheme depending on $V(c)$. The function V has a value 0 on the interior points.[1]

The periodic background color is then obtained by choosing, for each colour channel, $K_c > 0$ (with c intended as the colour channel) and by setting:

$$colour_c = \frac{255}{2 * K_d} * \left(1 - \cos\left(\frac{\log V}{K_c}\right)\right).$$

In this way, for each colour channel, we'll obtain values ranging between $\left[0, \frac{255}{K_d}\right]$. By lowering the range (i.e., by using $K_d > 1$), we can include also normal map effect





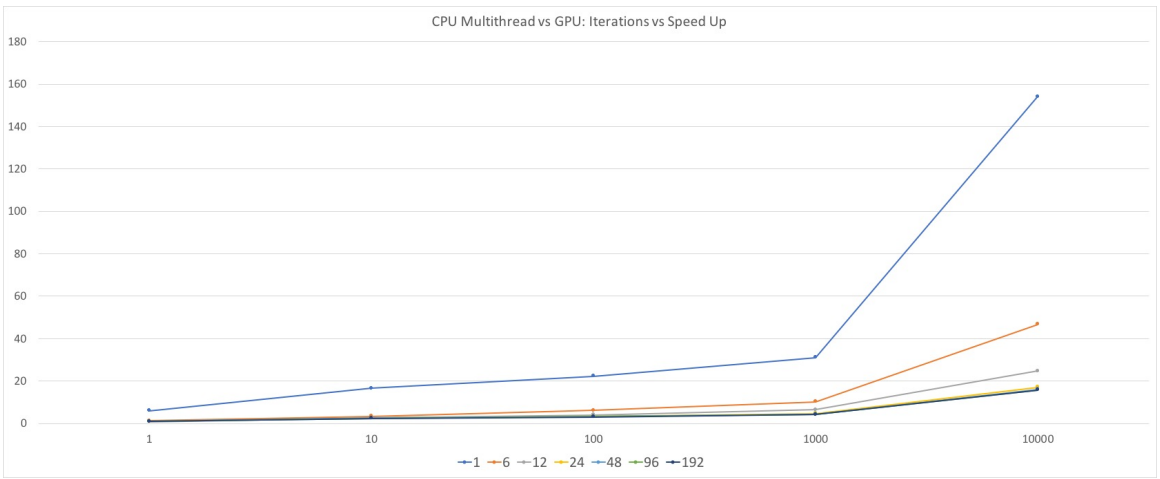


Figure 2.7: Different views of the same benchmark done to compare CPU vs GPU execution time. As for the other benchmarks, the settings are: 4000x2000, Zoom 0.5, X -0.6, Y 0.0, R 1000, no approximation.

and stripe average without having problems with over-saturated colors (i.e. white background). If $K_d < 0$, it is possible subtract the colour, which is useful on white backgrounds. Some examples as shown in figure 3.1.

This technique in extreme zooms allows to have interesting results, showing filaments and spirals, as illustrated in picture 3.2.

3.2 Exterior Distance Estimation

It's possible to calculate the distance between a point c and the Mandelbrot set as:

$$d = \lim_{n\to\infty} 2 \cdot \frac{|f_c^n(z_0)| \cdot \ln |f_c^n(z_0)|}{|\frac{\partial}{\partial c} f_c^n(z_0)|}$$

As seen before for the case of the potential, as soon as R is big enough and $|z| > R$, it's possible to calculate a good approximation of d.

This technique can be used both for colouring (choosing a colour depending on the distance between the point and the set) and (as a direct consequence) also to draw the boundaries of the set.[1]

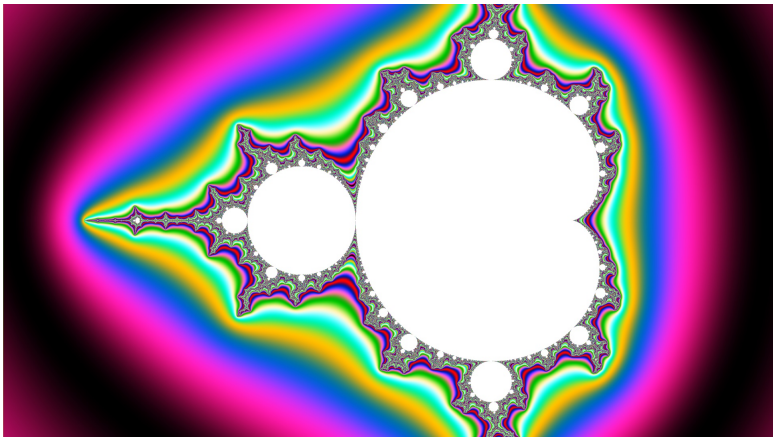
The border is drawn on top of all the other effects, and it's faded to smoothly join with the original background of the set. By setting it thick enough, it's also possible to obtain a sort of glow (if light) or shadow (if dark) effect. Possible examples are shown in figure 3.3

3.3 Interior Distance Estimation

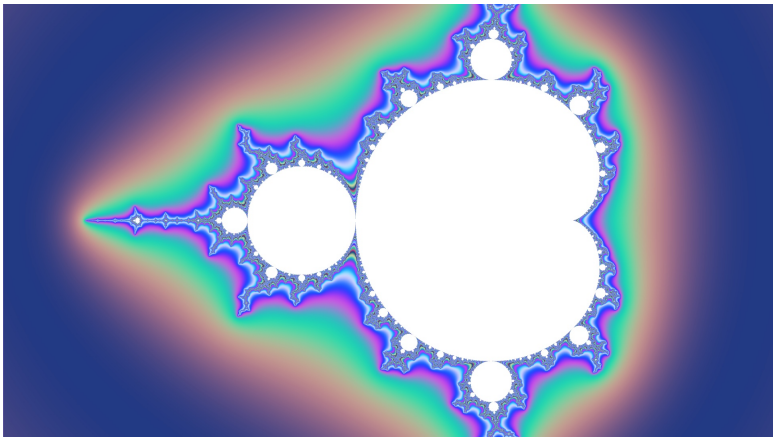
It is also possible to estimate the distance of an inner point to the boundary of the Mandelbrot set. This distance is calculated as:

$$d = \frac{1 - \left|\frac{\partial}{\partial z} f_c^n(z_0)\right|^2}{\left|\frac{\partial}{\partial c}\frac{\partial}{\partial z} f_c^n(z_0) + \frac{\partial}{\partial z}\frac{\partial}{\partial z} f_c^n(z_0) \frac{\frac{\partial}{\partial c} f_c^n(z_0)}{1 - \frac{\partial}{\partial z} f_c^n(z_0)}\right|}$$

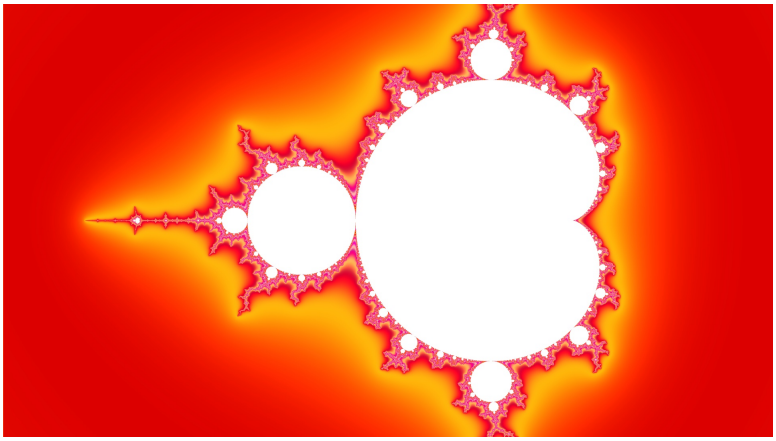
As always, it's possible to obtain a good approximation with a high value of maximum iterations.[8] One particular detail with interior distance estimation is that if approximation techniques are applied, artifacts start appearing in the center of each bulb, and the gradient start to be dependent on the bulb size (bulb period). An example is illustrated in figure 3.4: Inside of the main bulb, with $\epsilon = 0.0001$, periodic circles start appearing in the center, and the smaller bulbs have no gradient. In the image without approximation, instead, the gradient is evenly distributed in all the bulbs. The presence of those artifacts is intrinsically related to the early-escaped number of iterations provided by the approximation condition.



(a) $K_r = 0.2$, $K_g = 0.9$, $K_b = 0.3$, $K_d = 1$, background: #00000



(b) $K_r = 0.4$, $K_g = 0.7$, $K_b = 2.1$, $K_d = 0.7$, background: #223983



(c) $K_r = 1.1$, $K_g = 1$, $K_b = 6.1$, $K_d = 1.4$, background: #dd0000

Figure 3.1: Different configurations of periodic colouring

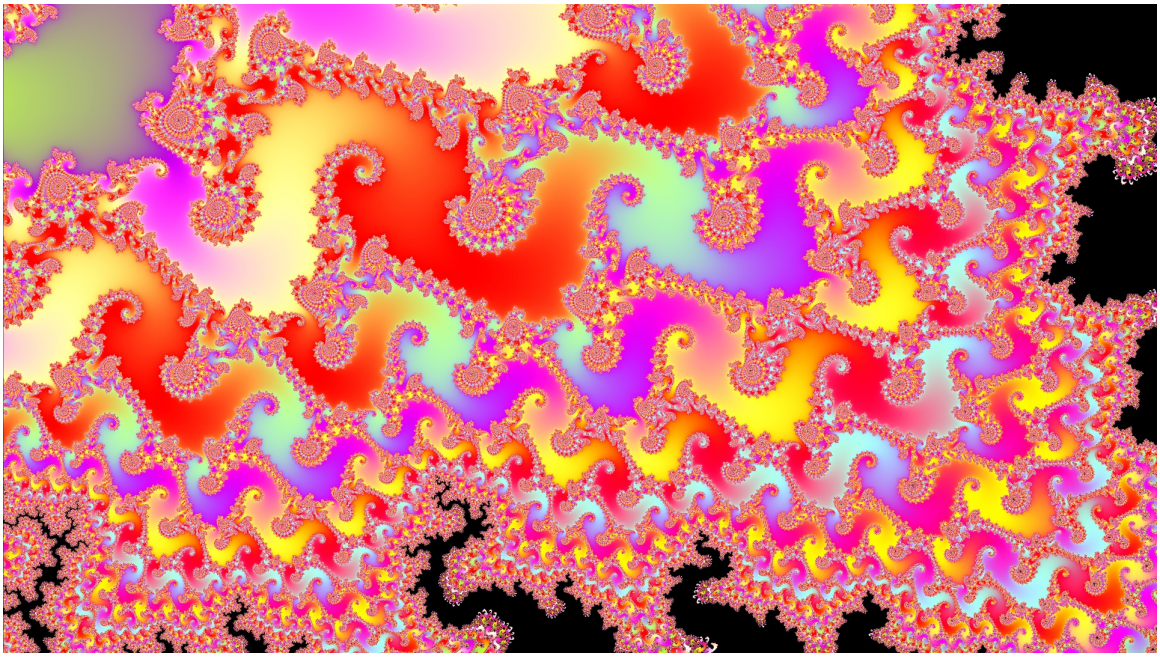


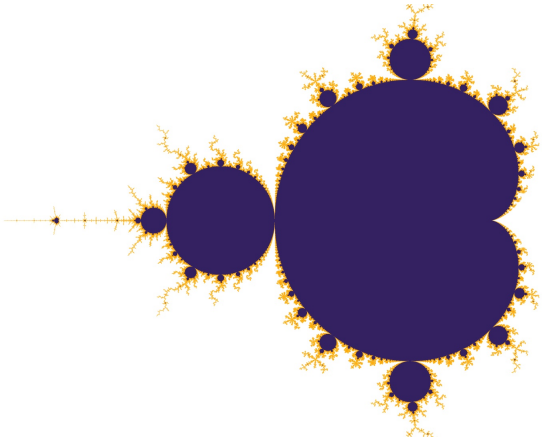
Figure 3.2: Spirals and filaments (X -0.7436442, Y -0.1318262, Zoom 2469765.75)

4 Other graphical effects

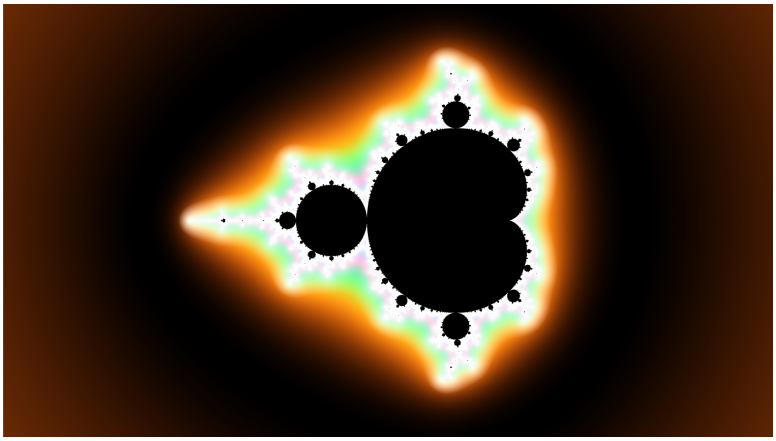
There are several other graphical improvements that can be applied on top of all the previously mentioned techniques. In particular, two different particular techniques were implemented, creating even more interesting results.

4.1 Stripe Average Colouring

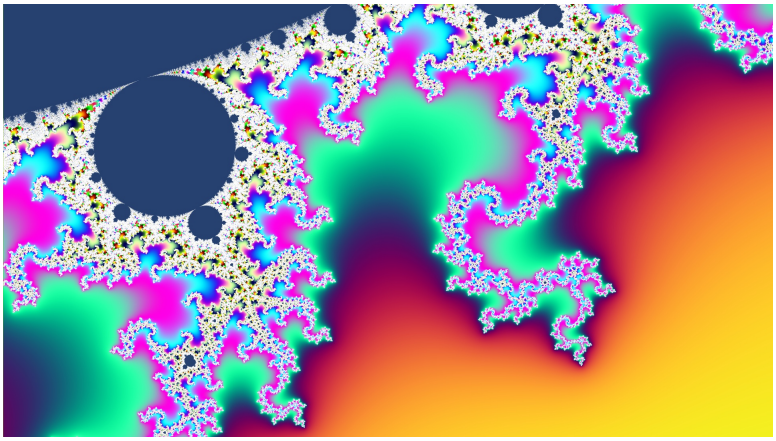
Stripe Average Colouring (or also called Stripe Average Method) is a technique which shows the trajectory of the points outside the Mandelbrot set. For each point, its orbit is calculated and, by taking the arithmetic mean of this series, a specific color is assigned consequently.[3] In particular, the Processing code found in Wikimedia - Mandelbrot set - Stripe Average Coloring (link) was used as a reference. Some examples are shown in figure 4.1. This effect is applied on top of all the others effects. In order to obtain lighter or darker colours, the RGB color was temporary converted into HSL or HSV format, and then, after changing the Lightness or Value component of the color, converted back to RGB format. This color technique produces gradients within the same hue, producing better results than other (easier) techniques.



(a) thickness = 0.001, color #ffaa00

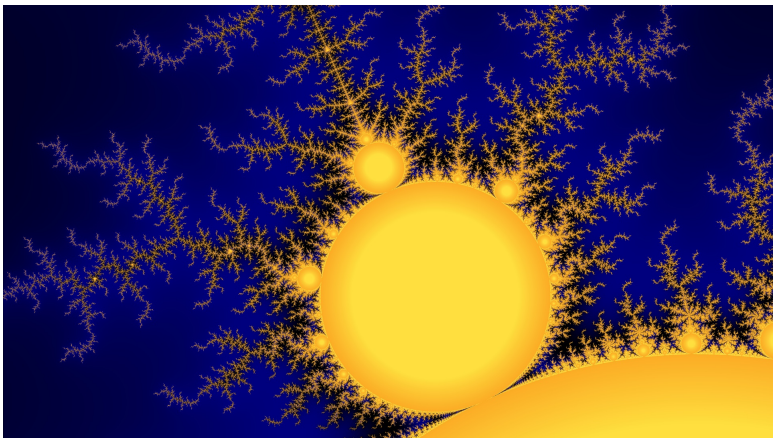


(b) thickness = 0.1, color #ffffff, with periodic background

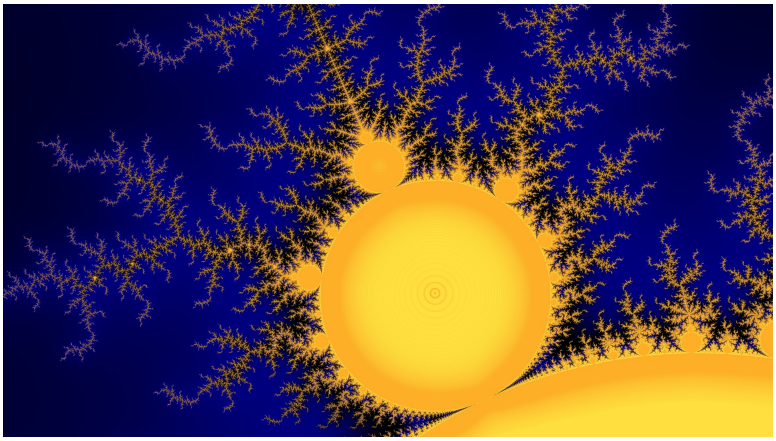


(c) thickness = 0.001, color #ffffff, with periodic background

Figure 3.3: Different border configurations

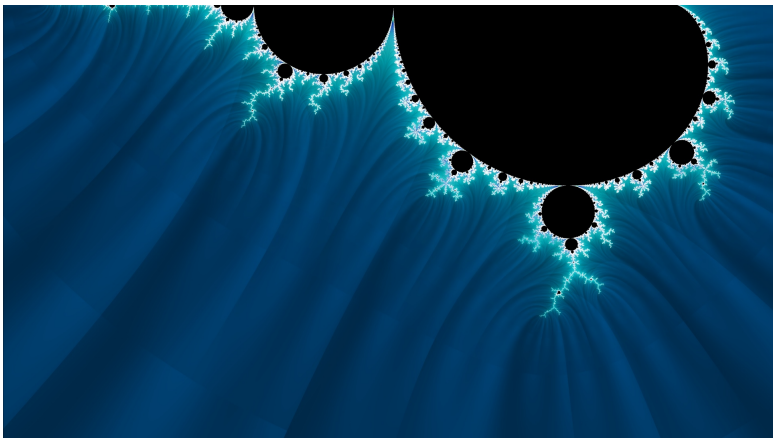


(a) No approximation

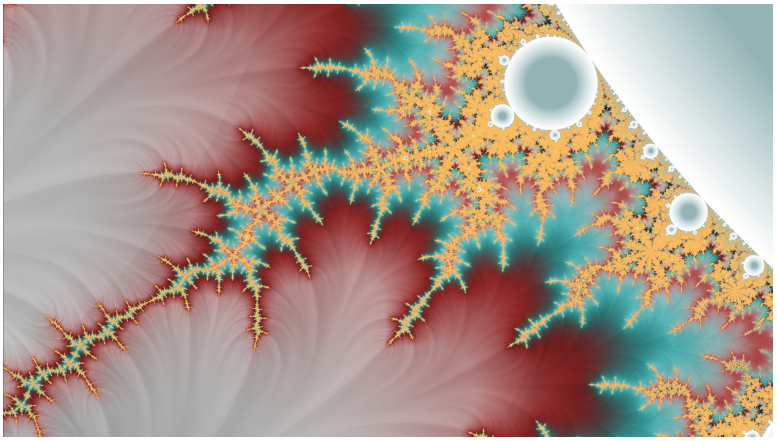


(b) Approximation $\epsilon = 0.0001$

Figure 3.4: Interior Distance Estimation

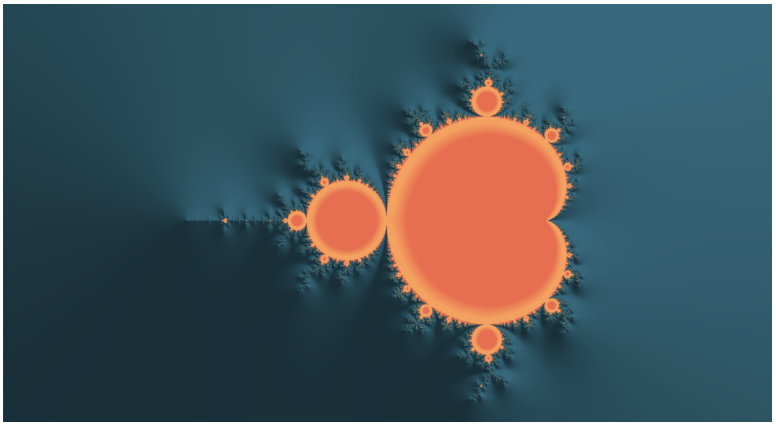


(a)

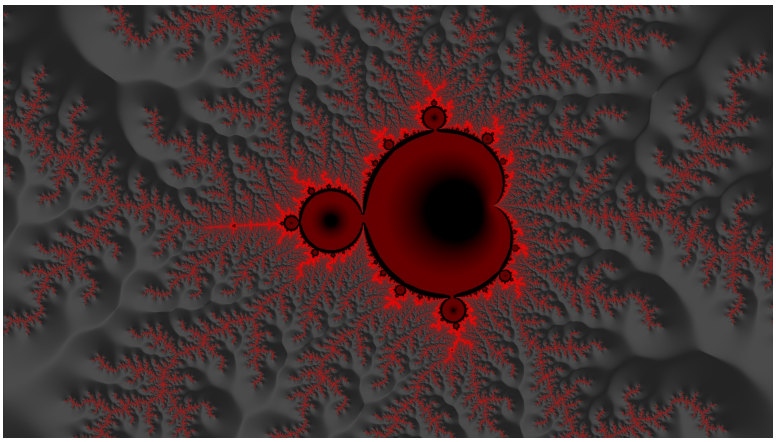


(b)

Figure 4.1: Stripe Average Colouring



(a) 45° angle



(b) 225° angle

Figure 4.2: Normal map effect

4.2 Normal map effect

Finally, a normal map effect was introduced, in particular the Milnor's distance estimator variant. This effect adds an interesting 3-dimensional effect to the fractal. This techniques allow to change the position of the light in its angle and distance (varying the strength of the shadows).[1] As for the case of Stripe Average Colouring, in order to obtain lighter or darker colours, the color was converted to HSL/HSV and then back to RGB. An example with two different light's angle is shown in figure 4.2.

5 OpenCV: Visualization

In order to visualize the image in real time, OpenCV was introduced. Since OpenCV is not compilable with the nvcc compiler, the code was split in two: A pure "host" code, which uses OpenCL for visualizing the image, and a CUDA code, which manages the fractal generation. Those two sources are compiled separately into object files (one with nvcc, one with g++), and then the object files are linked together by g++. A specific Makefile was introduced in order to simplify the process of compilation.

The image generated by the GPU is converted by OpenCV into an image, which is then down-sampled to the original desired resolution (the image is rendered at twice the resolution to introduce anti-aliasing due to down-sampling). By the specific functionalities offered by OpenCV, the image is then visualized inside of a window, accepting keyboard inputs for changing in real-time the settings of the generation and the position in the fractal space. The image can also be saved in JPEG format by OpenCV's *imwrite* function.

6 Possible improvements

There are several improvements that can be applied in order to speed up even more the execution time and improve the user friendliness of the application.

One possible technique is to render intermediate images of the fractal, rendered at a lower resolution or only partially (e.g., the not-yet calculated pixels can be interpolated from the neighbouring ones). In this way the user can see the generation state instead of having to wait for the full render to finish.

Another improvement could be the introduction of a graphical interface. Actually, the input keys are mapped pretty much in a nonsensical way, so every time that the user needs to change a specific settings, he'll need to check the console to remember the key mapping.

There are also advanced mathematical techniques, such as those based on perturbation theory, which allows to speed up consistently the execution time.[8]

Currently, in order to allow the compiler to optimize the calculations as much as possible, the calculations done in complex space are done piece-by-piece, without any helper function. This created some difficulties in the management of variants of Mandelbrot set, for example the ones with different exponentiation ($f_c(z) = z^d + c$, with $d > 2$).[7] Even if the code for the generation of the fractal is currently capable of generating Mandelbrot with different exponentiation, the other effects are not. This

could bring also the possibility to implement Julia set generation[5], or Buddhabrot[4].

In general, there are also some other techniques that can be exploited to improve further more the aesthetics of the image: internal color by orb's period, internal colour by internal radial angle, internal rays, etc.[2]

Bibliography

[1] *Mandelbrot set - Techniques for computer generated pictures in complex dynamics*. URL: https://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set. (accessed: 20/12/2020).

[2] *Wikibooks: Fractals/Iterations in the complex plane/Mandelbrot set interior*. URL: https://en.wikibooks.org/wiki/Fractals/Iterations_in_the_complex_plane/Mandelbrot_set_interior. (accessed: 01/01/2021).

[3] *Wikibooks: Fractals/Iterations in the complex plane/stripeAC*. URL: https://en.wikibooks.org/wiki/Fractals/Iterations_in_the_complex_plane/stripeAC. (accessed: 01/01/2021).

[4] *Wikipedia: Buddhabrot*. URL: https://en.wikipedia.org/wiki/Buddhabrot. (accessed: 01/01/2021).

[5] *Wikipedia: Julia Set*. URL: https://en.wikipedia.org/wiki/Julia_set. (accessed: 01/01/2021).

[6] *Wikipedia: Mandelbrot Set*. URL: https://en.wikipedia.org/wiki/Mandelbrot_set. (accessed: 20/12/2020).

[7] *Wikipedia: Mandelbrot set - Multibrot*. URL: https://en.wikipedia.org/wiki/Mandelbrot_set#Multibrot_sets. (accessed: 01/01/2021).

[8] *Wikipedia: Plotting algorithms for the Mandelbrot set*. URL: https://en.wikipedia.org/wiki/Plotting_algorithms_for_the_Mandelbrot_set. (accessed: 20/12/2020).