McGill University

ECSE 420:

Parallel Computing Project

Fractal Generation and OpenCL

Author:
Thomas Raynaud de Lage

December 14, 2015

Table of Contents

1. Introduction:

The goal of this project is to investigate the capabilities of OpenCL and determine how it compares with other techniques. We will focus on how to obtain a speedup, and what are its strengths and weaknesses. In order to do so, we generate fractal images using the C programming language and compare the relative speedups of the OpenCL (CPU and GPU), POSIX Threads, and sequential versions.

2. Background:

A fractal object is any mathematical set that exhibits. They appear everywhere in nature [1] , and find applications in a broad range of subjects from file compression to fluid mechanics .
In order to test OpenCL and apply parallel computing, I will focus on 2 famous fractal sets: the Mandelbrot Set and the Julia Set.

3. Methodology:

I conducted 3 experiments to test the capabilities of OpenCL and achievable speedups. For all experiments, I used my own computer a MacBook Pro Mid-2012 with:
CPU: Intel Core i7 (4 cores, 2 threads per core)
GPU: NVIDIA GeForce GT 650M (384 CUDA cores)

3.1. Experiment 1:

The first experiment was to create a Mandelbrot set image of size 2048×2048 pixels (see FIgure 1). The Mandelbrot set is defined by the following recursion:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

$$\forall c \in \mathbb{C}$$

for which the recursion does not diverge.

In practice I restricted the the complex points of interests in the range $[-2.5, 1.5]$ for the real part and $[-2.0, 2.0]$ for the imaginary part. Each pixel is then mapped to a complex point.
In order to test for divergence, I set a maximum number of iterations to 300, and if after these 300 iterations the recursion is not superior or equal to 4, we consider the complex point c belongs to the Mandelbrot set. Figure 1 is an output of my program, the black pixels (or their complex equivalents) belong in the Mandelbrot set (or at least the recursion does not diverge after 300 iterations at that point), the other colors represent the speed of divergence, from white being the slowest to diverge to dark red being the fastest (in terms of number of iterations).

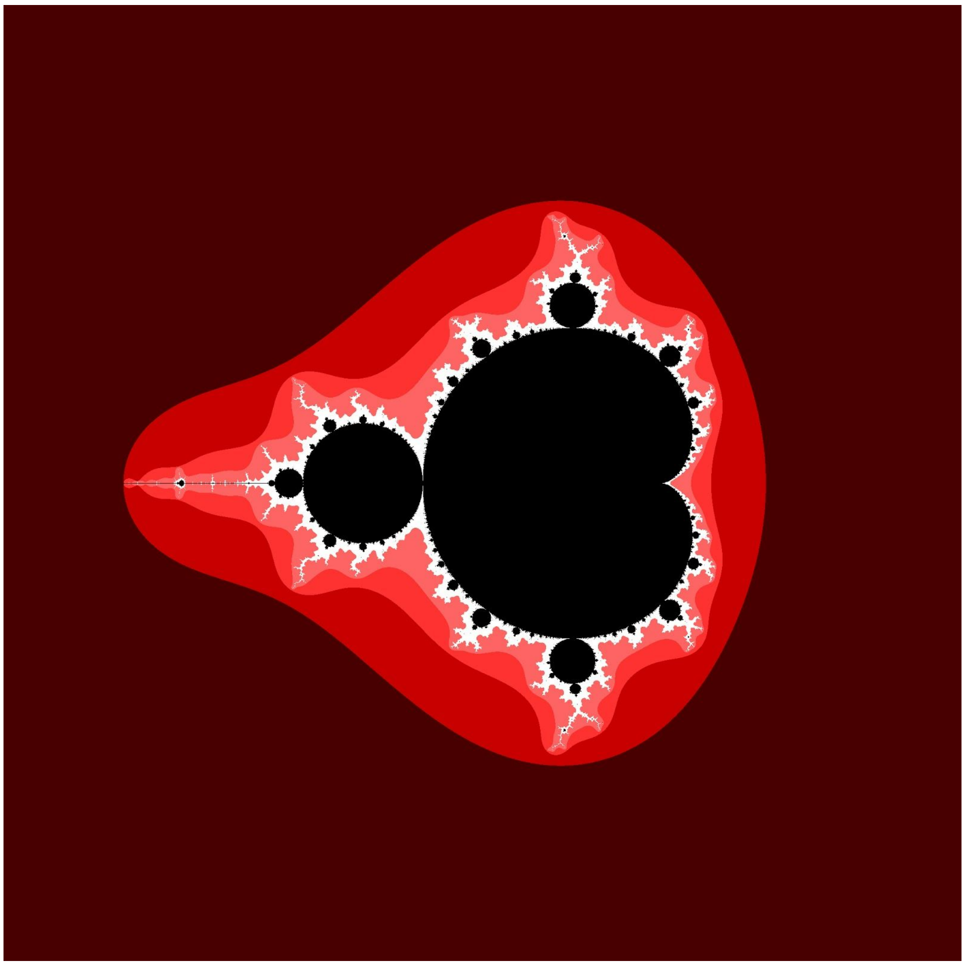


Figure 1: Generated Mandelbrot Set image of size 2048 × 2048 pixels with max iterations set to 300

I first ran a sequential version 100 times (see `sequential_mandelbrot.c`) to obtain the purely sequential execution time (see Table 1).

File	Size (pixels)	Maximum number of iterations	Total Run Time (100 runs)(s)
sequential_mandelbrot.c	2048*2048	300	176

Table 1: Characteristics of the sequential generation of the Mandelbrot set

Then, I ran the Pthread version 100 times (see `pthread_mandelbrot.c`), with different numbers of threads (see Table 2). I decided to parallelize the work over the width, each thread taking care of a portion of the width of the image. Interestingly, the optimal speedup was not obtained when running one thread per core (in this case 512 threads total instead of the expected 8). This is due to the fact that each thread will have a very different amount of work to perform. For example, if a thread has a portion of width completely to the left of the image, all of the complex numbers in this range diverge after 1 iteration, and thus that particular thread will finish much faster than a thread with a portion of the width in the middle of the image where a lot of number do not diverge after the maximum number of iterations (in this case 300).

File	Size (pixels)	Maximum number of iterations	Number of threads	Columns per thread	Total Run Time (100 runs)(s)
pthreads_mandelbrot.c	2048*2048	300	1	2048	176
pthreads_mandelbrot.c	2048*2048	300	2	1024	137
pthreads_mandelbrot.c	2048*2048	300	4	512	137
pthreads_mandelbrot.c	2048*2048	300	8	256	101
pthreads_mandelbrot.c	2048*2048	300	16	128	78
pthreads_mandelbrot.c	2048*2048	300	32	64	64
pthreads_mandelbrot.c	2048*2048	300	64	32	62
pthreads_mandelbrot.c	2048*2048	300	128	16	61
pthreads_mandelbrot.c	2048*2048	300	256	8	58
pthreads_mandelbrot.c	2048*2048	300	512	4	59
pthreads_mandelbrot.c	2048*2048	300	1024	2	60
pthreads_mandelbrot.c	2048*2048	300	2048	1	65

Table 2: Characteristics of the Pthread generation of the Mandelbrot set

I then ran an OpenCL version 100 times (see `opencl_mandelbrot.c` and `mykernel.cl` in the mandelbrot folder) on my GPU with different numbers of work groups (see Table 3). In practice, I actually defined the size of each work group which has to be a factor of the total size (in this case 2048), so for example if we set each work group size to 4, there will be 512 groups. There is also another option, which is to pass 0 as the work group size, the size is then computed automatically based on the OpenCL device used and the kernel, however this approach does not always give the optimal result (in this case it yields 2 groups which is far from optimal, see Table 3).
There are also restrictions on the maximum size of the work group we can use, depending on the device. We can query the OpenCL devices for relevant information (I have compiled some of the important queries I have found in `device_info.c`). In the case of my GPU the maximum size of a work group is 1024 (see Appendix 6.1 for the output of `device_info.c` on my computer).
In some cases, with too few work groups I found that we lose part of the image (See Appendix 6.2 Figure 5). I am assuming this happens because one or several threads are performing too much work (the loss always occur on columns with the highest number of points that reach the maximum number of iterations).

File	Size (pixels)	Maximum number of iterations	Number of working groups	Columns per work group	Total Run Time (100 runs)(s)	Status
opencl_mandelbrot.c	2048*2048	300	-	-	-	-
opencl_mandelbrot.c	2048*2048	300	2	1024	805	incomplete
opencl_mandelbrot.c	2048*2048	300	4	512	857	incomplete
opencl_mandelbrot.c	2048*2048	300	8	256	815	incomplete
opencl_mandelbrot.c	2048*2048	300	16	128	685	complete
opencl_mandelbrot.c	2048*2048	300	32	64	385	complete
opencl_mandelbrot.c	2048*2048	300	64	32	210	complete
opencl_mandelbrot.c	2048*2048	300	128	16	165	complete
opencl_mandelbrot.c	2048*2048	300	256	8	135	complete
opencl_mandelbrot.c	2048*2048	300	512	4	127	complete
opencl_mandelbrot.c	2048*2048	300	1024	2	127	complete
opencl_mandelbrot.c	2048*2048	300	2048	1	127	complete

Table 3: Characteristics of the OpenCL GPU generation of the Mandelbrot set

Finally, as one of the advantages of OpenCL is its portability, I tested the kernel code on my CPU as well (see Table 4). The only value that was accepted for the size of a work group was 1 (2048 groups).

File	Size (pixels)	Maximum number of iterations	Number of working groups	Columns per work group	Total Run Time (100 runs)(s)	Status
opencl_mandelbrot.c	2048*2048	300	2048	1	50	complete

Table 4: Characteristics of the OpenCL CPU generation of the Mandelbrot set

3.2. Experiment 2:

I conducted a similar experiment to Experiment 1, this time creating a Julia set image of size 2048×2048 pixels (see Figure 2). A Julia set is similar to a Mandelbrot set, except it is initialized with *c* and incremented with a constant complex number *k*. It is defined by the following recursion.

$$\begin{cases} z_0 = c \\ z_{n+1} = z_n^2 + k \end{cases}$$

$$\forall c \in \mathbb{C}$$

$$k \in \mathbb{C}$$

for which the recursion does not diverge.

Since k is a constant complex number, there is an infinite number of Julia sets, in my case I used $k = 0.285 + i0.013$.

In practice I restricted the the complex points of interests in the range $[-2.0, 2.0]$ for the real part and $[-2.0, 2.0]$ for the imaginary part. Each pixel is then mapped to a complex point.
In order to test for divergence, I set a maximum number of iterations to 300, and if after these 300 iterations the recursion is not superior or equal to 4, we consider the complex point c belongs to the Julia set. Figure 2 is an output of my program, the black pixels (or their complex equivalents) belong in the Julia set (or at least the recursion does not diverge after 300 iterations at that point), the other colors represent the speed of divergence, from white being the slowest to diverge to dark blue being the fastest (in terms of number of iterations).

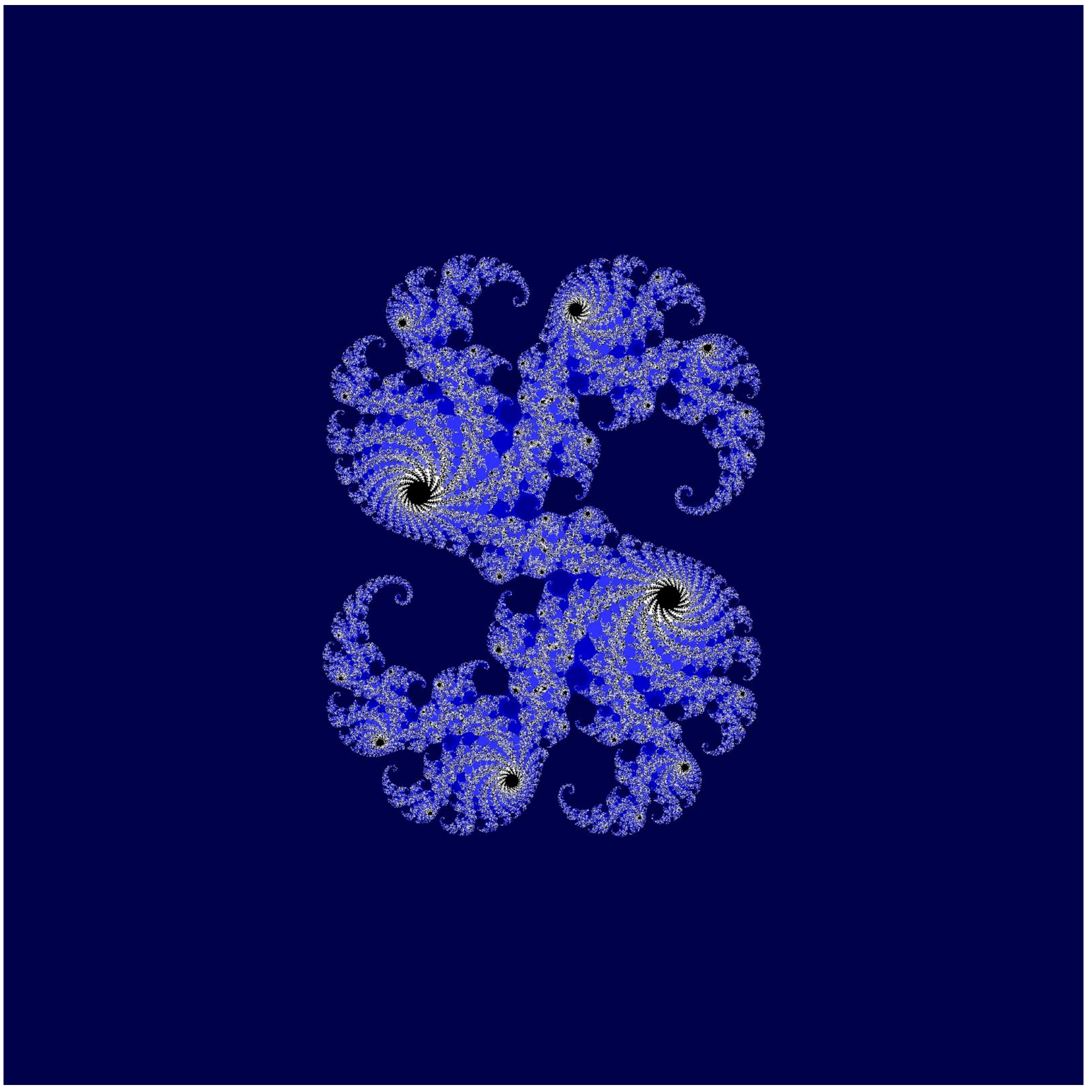


Figure 2: Generated Julia Set image of size 2048 × 2048 pixels with max iterations set to 300

I first ran a sequential version 100 times (see `sequential_julia.c`) to obtain the purely sequential execution time(see Table 5).

File	Size (pixels)	Maximum number of iterations	Total Run Time (100 runs)(s)
sequential_julia.c	2048*2048	300	132

Table 5: Characteristics of the sequential generation of the Julia set

Then, I ran the Pthread version 100 times (see `pthread_mandelbrot.c`), with different numbers of threads (see Table 2).

File	Size (pixels)	Maximum number of iterations	Number of threads	Columns per thread	Total Run Time (100 runs)(s)
pthreads_julia.c	2048*2048	300	1	2048	129
pthreads_julia.c	2048*2048	300	2	1024	85
pthreads_julia.c	2048*2048	300	4	512	84
pthreads_julia.c	2048*2048	300	8	256	71
pthreads_julia.c	2048*2048	300	16	128	58
pthreads_julia.c	2048*2048	300	32	64	56
pthreads_julia.c	2048*2048	300	64	32	54
pthreads_julia.c	2048*2048	300	128	16	53
pthreads_julia.c	2048*2048	300	256	8	53
pthreads_julia.c	2048*2048	300	512	4	54
pthreads_julia.c	2048*2048	300	1024	2	55
pthreads_julia.c	2048*2048	300	2048	1	60

Table 6: Characteristics of the Pthread generation of the Julia set

I then ran an OpenCL version 100 times (see `opencl_julia.c` and `mykernel.cl` in the julia folder) on my GPU with different numbers of work groups (see Table 7).
As for the Mandelbrot Set image generation, with too few work groups I found that we lose part of the image (See Appendix 6.2 Figure 6).

File	Size (pixels)	Maximum number of iterations	Number of working groups	Columns per work group	Total Run Time (100 runs)(s)	Status
opencl_julia.c	2048*2048	300	-	-	-	-
opencl_julia.c	2048*2048	300	2	1024	858	incomplete
opencl_julia.c	2048*2048	300	4	512	905	incomplete
opencl_julia.c	2048*2048	300	8	256	885	incomplete
opencl_julia.c	2048*2048	300	16	128	688	complete
opencl_julia.c	2048*2048	300	32	64	372	complete
opencl_julia.c	2048*2048	300	64	32	215	complete
opencl_julia.c	2048*2048	300	128	16	172	complete
opencl_julia.c	2048*2048	300	256	8	148	complete
opencl_julia.c	2048*2048	300	512	4	125	complete
opencl_julia.c	2048*2048	300	1024	2	114	complete
opencl_julia.c	2048*2048	300	2048	1	109	complete

Table 7: Characteristics of the OpenCL GPU generation of the Julia set

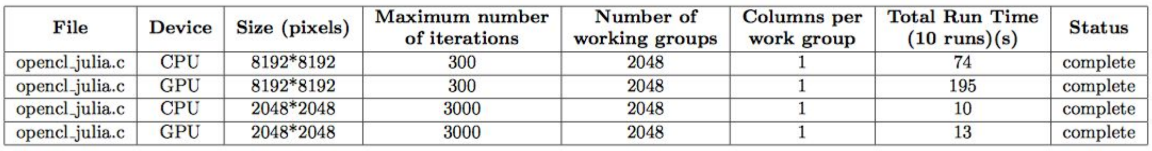
Finally,I tested the kernel code on my CPU as well (see Table 8). The only value that was accepted for the size of a work group was 1 (2048 groups).

File	Size (pixels)	Maximum number of iterations	Number of working groups	Columns per work group	Total Run Time (100 runs)(s)	Status
opencl_julia.c	2048*2048	300	2048	1	52	complete

Table 8: Characteristics of the OpenCL GPU generation of the Mandelbrot set

3.3. Experiment 3:

In this experiment I attempted to increase the size of the problem. Indeed, since the GPU access creates a lot of delay and overhead, the only way to obtain a speedup of the OpenCL GPU version against OpenCL CPU version is to increase the size enough to make up for that initial delay. I performed the tests compiled in Table 9.
I increased both the maximum number of iterations and the size of the image in the `opencl_julia.c` file to see if I could obtain a speedup, or at least come closer.



File	Device	Size (pixels)	Maximum number of iterations	Number of working groups	Columns per work group	Total Run Time (10 runs)(s)	Status
opencl_julia.c	CPU	8192*8192	300	2048	1	74	complete
opencl_julia.c	GPU	8192*8192	300	2048	1	195	complete
opencl_julia.c	CPU	2048*2048	3000	2048	1	10	complete
opencl_julia.c	GPU	2048*2048	3000	2048	1	13	complete

Table 9: Test results for Experiment 3

4. Results and Analysis:

4.1.Experiment 1 Results:

From Experiment 1 and Figure 3 we can see that Pthread, OpenCL GPU, and OpenCL CPU achieve a speedup. For the Pthread version the maximum speedup obtained is 3.03,for the OpenCL version the maximum speedup achieved is 1.385, and for the OpenCL GPU the maximum speedup is 3.52.

We expected the speedup of the OpenCL GPU version to be higher as there are more (CUDA) cores, but there are a number of reasons that could explain this.

There are a number of limitations to the speed we can achieve using OpenCL on a GPU. FIrst of all Memory Copying from the source to the kernel executing on the GPU introduces a latency
Interestingly, the OpenCL CPU version outperforms the Pthread version. This must be due to fact that the threads are more lightweight in OpenCL.

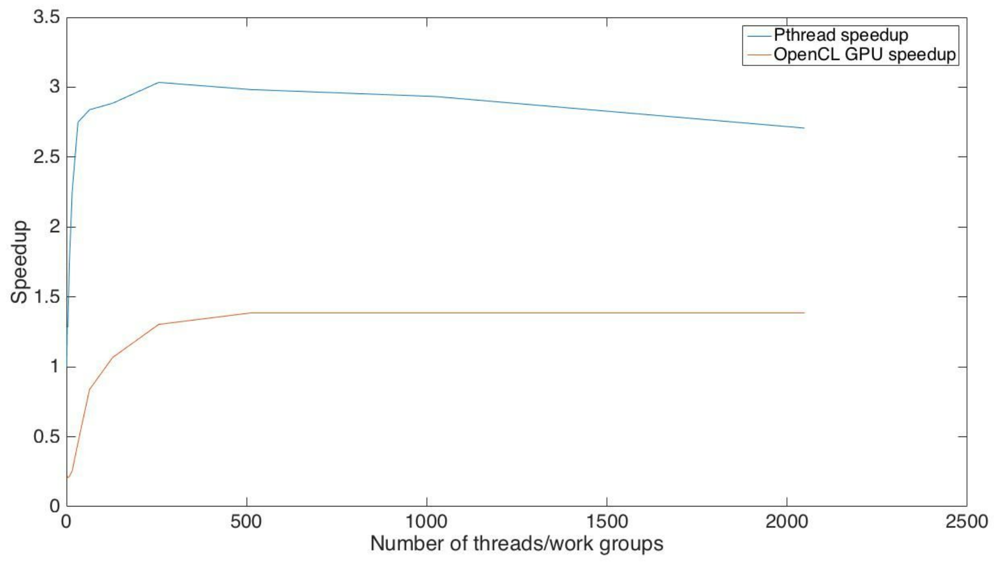


Figure 3: Speedup generating the Mandelbrot Set image of size 2048 × 2048 pixels with max iterations set to 300

4.2. Experiment 2 Results:

We obtain similar results to Experiment 1, except the speedups are lower (see Figure 4). For the Pthread version the maximum speedup obtained is 2.491,for the OpenCL version the maximum speedup achieved is 1.211, and for the OpenCL GPU the maximum speedup is 2.481. This must be due to the fact that the set is more condensed and so there are a lot of inefficient threads that perform very little work.

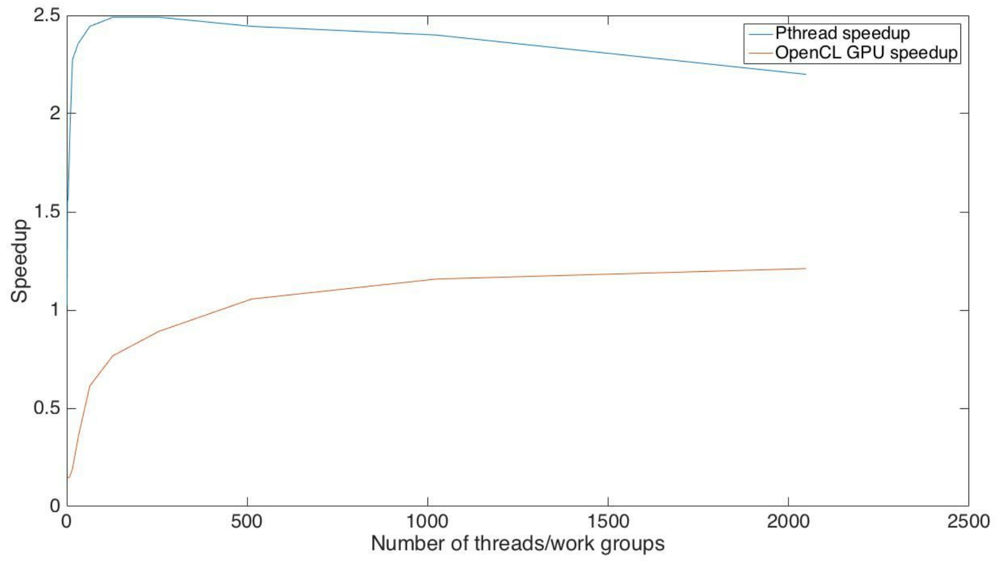


Figure 4: Speedup generating the Julia Set image of size 2048 × 2048 pixels with max iterations set to 300

4.3. Experiment 3 Results:

When I attempted to to increase the size of the problem through size in pixels of the image I did not get a speedup, in fact the gap seemed to widen. I am assuming this must have to do with the number of compute units of the devices. My CPU has 8 compute units whereas my GPU only has 2. It could also be that I require a far larger image to notice some speedup, but as the memory on my GPU is far more limited than my CPU I could end up losing parts of the image again.

5. Conclusion:

OpenCL shows a lot of promise for the future of Parallel Computing. It allows to run kernel code on a variety of OpenCL capable devices including CPUs, GPUs, and FPGAs. This an advantage over its main competitor for GPU competition CUDA (which only works with Nvidia GPUs), as the parallel part can be written only once and will run on different devices (however, the optimization may vary).

Nonetheless, one must carefully choose the tasks that should be run on the GPU (or other) and know how to optimize them. This requires proper knowledge of the different available devices, such as the maximum size of a work group or the maximum work item size, as well as the architecture of the system.
Another issue is the lack of documentation and since many issues encountered are specific to a the configuration of a single machine it can be hard to find resources to solve the a problem.

6. Appendix:

6.1. Output of `device_info.c`:

```
Number of platforms:   1

Platform:        0

      Platform Vendor:  Apple
      Number of devices:     3

      Device: 0
            Name:                        Intel(R) Core(TM) i7-3615QM CPU
@ 2.30GHz
            Vendor:                      Intel
            Available:                   Yes
```

```
        Compute Units:               8
        Clock Frequency:             2300 mHz
        Global Memory:               8192 mb
        Max Allocatable Memory:      2048 mb
        Local Memory:                32768 kb
        device max work group size : 1024
        device max work item dimensions:3
        device max work item sizes:  1024 / 1 / 1

  Device: 1
        Name:                        HD Graphics 4000
        Vendor:                      Intel
        Available:                   Yes
        Compute Units:               16
        Clock Frequency:             1200 mHz
        Global Memory:               1024 mb
        Max Allocatable Memory:      256 mb
        Local Memory:                65536 kb
        device max work group size : 512
        device max work item dimensions:3
        device max work item sizes:  1024 / 1 / 1

  Device: 2
        Name:                  GeForce GT 650M
        Vendor:                      NVIDIA
        Available:                   Yes
        Compute Units:               2
        Clock Frequency:             774 mHz
        Global Memory:               512 mb
        Max Allocatable Memory:      128 mb
        Local Memory:                49152 kb
        device max work group size : 1024
        device max work item dimensions:3
        device max work item sizes:  1024 / 1 / 1
```

Note: For some reason I was never able to access my Intel HD 4000, although it says it is available. This may be an issue with Apple or Mavericks or even a specificity on the type of kernel it accepts, I am not sure.

6.2. Incomplete Images:

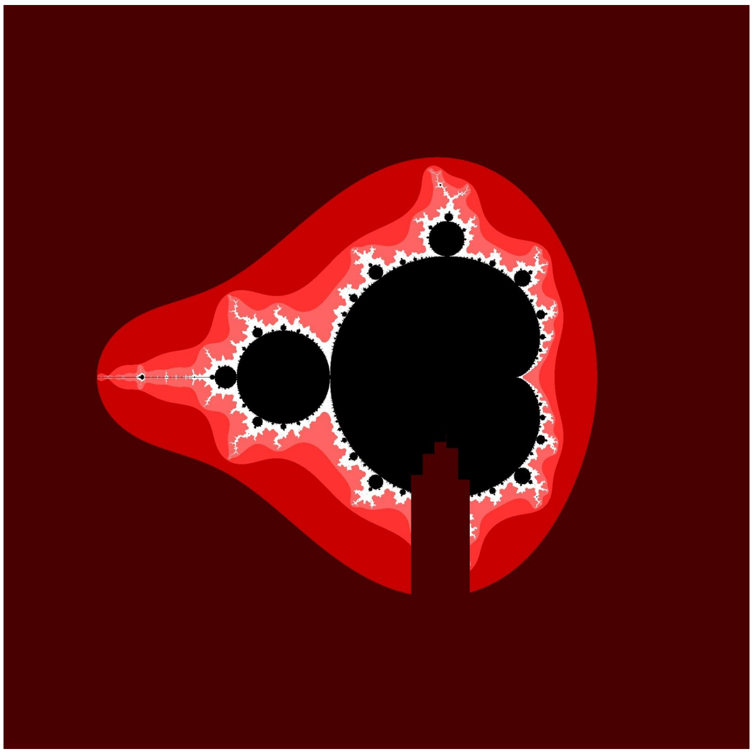


Figure 5: Generated Mandelbrot Set image of size 2048 × 2048 pixels with max iterations set to 300 using OpenCL GPU with 8 working groups

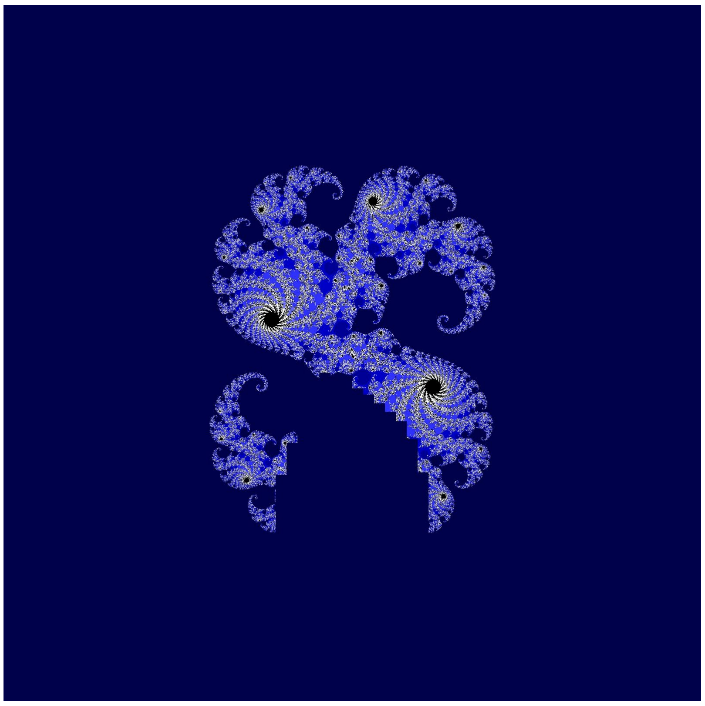


Figure 6: Generated Julia Set image of size 2048 × 2048 pixels with max iterations set to 300 using OpenCL GPU with 8 working groups

7. References

[1] B. B. Mandelbrot, “The Fractal Geometry of Nature,” *Am. J. Phys.*, vol. 51, no. 3, p. 286, 1983.