Developing a parallel version of Mandelbrot using OpenCL

Luciano Araújo Dourado Filho[1], **Almir Moreira da Silva Neto**[1]

[1] PGCC - Computer Science Postgraduate Program
Universidade Estadual de Feira de Santana
Bahia, Brazil

lucianoadfilho@gmail.com[1],

almirneto338@gmail.com[1]

1. Introduction

This report presents a comparative analysis of performance for the computation and visualization of the Mandelbrot set using the escape-time algorithm. Our experiments covered 2 distinct platforms and was done using the Open Computing Language (OpenCL) API. The OpenCL language provides a toolkit for developing functions that run in parallel on multiple computing units. It allows for programming heterogeneous applications (multiple device platforms) in a standardized way [Tompson and Schlachter 2012].

The work is presented as follows: Section 1.1 presents the escape-time algorithm, Section 2 introduces the OpenCL platform and details important concepts related to it. Section 3 presents the methodology and finally, Sections 3.1,4 and 5, presents the experiments, results and conclusion, respectively.

1.1. Escape-Time Algorithm

The Mandelbrot set is built by iterating over the function:

$$z_{n+1} = z_n^2 + c \tag{1}$$

, where $c(x, y) = x + iy$, is a complex number mapped from pixel coordinates (x, y) into the complex domain. If we establish $x_{min}, x_{max}, y_{min}, y_{max}$: the horizontal and vertical axes lengths for the complex plane, a $width$ and $height$ for the output image resolution, we can initialize $z = 0$ and iteratively update it through the formula above for a predetermined number of iterations $maxIter$. The process then consists of checking whether the sequence remains bounded or escapes to infinity i.e., $|z| > 2$, and then coloring each pixel proportionally to the number of iterations taken to escape. A more concise description of the algorithm is described in the Algorithm 1 and will be presented in more detail in the following sections.

Algorithm 1 Escape Time Algorithm for Mandelbrot Set

procedure COMPUTEMANDELBROT($x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}$, width, height, maxIter)
  for $i = 0$ to width $- 1$ **do**
    for $j = 0$ to height $- 1$ **do**
      $x \leftarrow x_{\text{min}} + i \times \frac{x_{\text{max}} - x_{\text{min}}}{\text{width}}$
      $y \leftarrow y_{\text{min}} + j \times \frac{y_{\text{max}} - y_{\text{min}}}{\text{height}}$
      $c \leftarrow x + iy$
      $z \leftarrow 0$
      $n \leftarrow 0$
      while $|z| \leq 2$ and $n <$ maxIter **do**
        $z \leftarrow z^2 + c$
        $n \leftarrow n + 1$
      end while
      Store n as the color value for pixel (i, j)
    end for
  end for
end procedure

2. OpenCL

As mentioned in Section 1, the OpenCL paradigm enables the programming of parallel applications in a multiple device environment through the execution of kernel functions. More precisely, in the OpenCL architecture, the kernels corresponds to functions written to run on a computing device e.g., graphics processing unit (GPU), to be executed in parallel as multiple work-items. In other words, computing devices are composed by compute units, which in turn are composed by processing elements (PE), in which the kernels are executed (Figure 1) [The Khronos Group 2020, Tompson and Schlachter 2012].

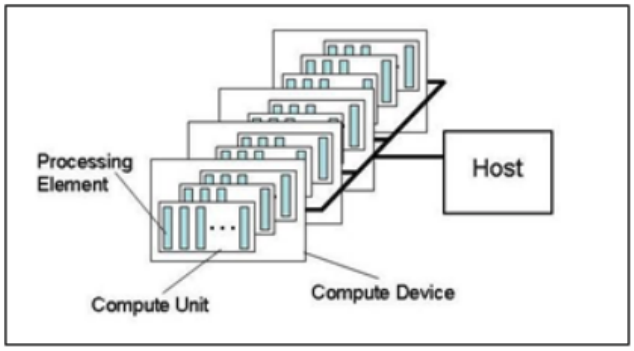


Figure 1. OpenCL Platform Model[1]

2.1. Execution Model

In the OpenCL execution model (Figure 1), the host (mostly the CPU) can query, select and manage compute devices for executing kernel functions [Tompson and Schlachter 2012]. To that end, it specifies a N-dimensional

[1] https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

computation domain (e.g., two-dimensional for images), in which each (N-dimensional) index corresponds to a work-item. The work-items on the other hand, are stacked into work-groups which allows for the simultaneous execution of multiple work-items in a parallel way. This model allows for defining image block operations as work-groups defined over pixel-wise operations (work-items). As we will see further, this will be perfectly suited to the computation of the Mandelbrot set through the Escape-time algorithm in a more efficient way. Section 1.1 presents the Escape-time algorithm in detail. Although the algorithm may appear to be sequential, due to the recursive nature of updating z, as it operates at pixel level, it can be optimized to run in a multi-thread parallel environment, by parallelizing pixel-wise computations.

3. Method

As mentioned earlier in Sections 1 and 1.1, the escape-time algorithm can be parallelized to perform simultaneous pixel-wise operations. In terms of OpenCL language, that amounts down to defining a kernel to perform the operations illustrated in Equation 1. By doing so, every work-item will execute this recursive update for the predefined amount of iterations (or until convergence) in an independent way and each work-group will operate over a block of pixels hence enabling faster computation.

The work-group size is subject to limitations constrained by the corresponding device that is being utilized. The work-group size for instance defines the maximum amount of concurrent threads that can be executed by the OpenCL kernel across all dimensions. In that sense, this hardware limitation constraints the number of work-items that can be executed concurrently. If the work-group size is 1024, for example, it means that each work group can at most, process a square block of 32x32 pixels.

3.1. Experiments

We have performed several experiments logging execution time for each parameter combination. We have fixed the image size to a resolution of 8K, meaning output image have 7680×4320, columns and rows respectively, resulting in more than 3 millions pixels. For each execution we evaluated different configurations of maximum amount of iterations ranging from 10000 to 1000000 with a step size of 10000. For all configurations, we measured only the execution time of the Mandelbrot set computation, excluding the time required to write the output image to disk.

3.1.1. Hardware Specifications

As depicted in [Tompson and Schlachter 2012], the specific definition of compute units is different depending on the hardware vendor. In NVIDIA hardware, (which are the ones employed in this work) they call compute units "stream multiprocessors" (SM). In every architecture, each SM can execute a limited number of threads per cycle. In NVIDIA's case, the amount of concurrent threads that each SM can execute is a multiple of what they call warps, which corresponds to groups of 32 threads. In our case, our experiments were conducted on two different NVIDIA computing devices: GTX1650 and A100SXM 80GB. Both devices have the capability of executing 64 threads per unit in groups of 2 warps (32 threads), the NVIDIA GTX1650 has 14 SMs (compute units) and the NVIDIA

A100SXM 80GB has 108 computing units, resulting in 896 and 6912 cores respectively. They both can operate on work-groups of size 1024 items e.g., blocks of 32x32 pixels. In terms of threads, the A100 can theoretically execute approximately 8 times more concurrent operations than the GTX1650.

A summary of the hardware specifications is described as follows:

- GTX1650:
  - Global memory size: 4 GB
  - Local memory size: 48 KB
  - Max compute units: 14
  - Num threads per unit: 64
  - Max work group size: 1024
  - OpenCL device version: OpenCL 3.0 CUDA
- A100SXM:
  - Global memory size: 80 GB
  - Local memory size: 48 KB
  - Max compute units: 108
  - Num threads per unit: 64
  - Max work group size: 1024
  - OpenCL device version: OpenCL 3.0 CUDA

3.1.2. System Specifications

All experiments were conducted on a system running Ubuntu, a Linux-based operating system. The system configuration is detailed below:

- Operating System: Ubuntu
- System Architecture: x86_64
- OpenCL Version: 3.0

3.1.3. Execution

We developed a simple script to automate the execution of our parallel implementations across different configurations of iteration values. The script systematically iterates over the number of iterations, ensuring that all combinations were executed in a controlled and consistent manner. Each execution was logged to a CSV file, recording the number of compute units of the device, the maximum number of iterations, and the corresponding execution time in seconds.

We utilized the event profiling function provided by the OpenCL API, which returns the time required to execute all kernels, in seconds. This approach offers a straightforward and portable method for measuring the execution time of all enqueued kernels.

4. Results and Discussions

Table 1 demonstrates the precise execution time for some experimental configuration. As mentioned before, we scaled the problem in terms of the number of iterations (N).

	Number of iterations			
Device	**10000**	**100000**	**500000**	**1000000**
GTX 1650	0.277460	2.566046	12.878640832	25.892726848
A100	0.050333	0.387145	1.921484800	3.834894336

Table 1. Execution time, in seconds, by device and maximum number of iterations

Table 1 shows only a sample of all performed executions by each device. In Figure 2 it is shown that A100, in orange, performs much faster than GTX 1650, in blue, for all maximum number of iterations configured. It is also shown that A100 is capable of handling a larger consuming workload in comparison with GTX 1650, since its curve increases a lot faster than A100's.

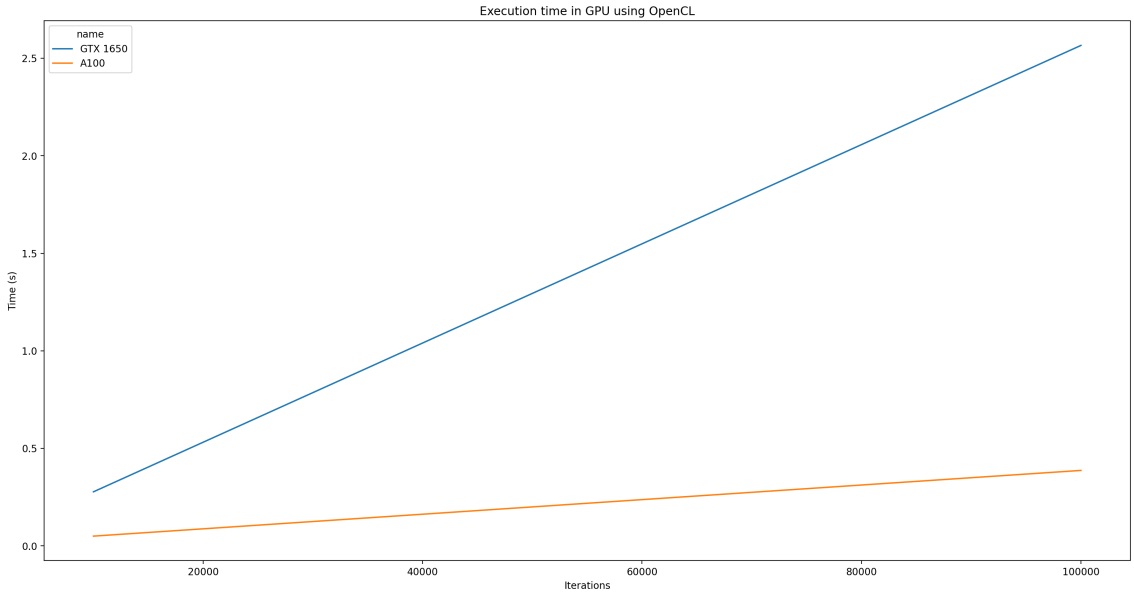


Figure 2. Execution time, in seconds, by device and number of iterations

To demonstrate the advantage of employing a heterogeneous architecture for processing a task with high workload, we have compared the execution time between a serial version and a homogeneous parallel version using OpenMP with 10 threads for execution. The execution time for each approach, using 10000 and 100000 maximum number of iterations, is depicted in Table 2.

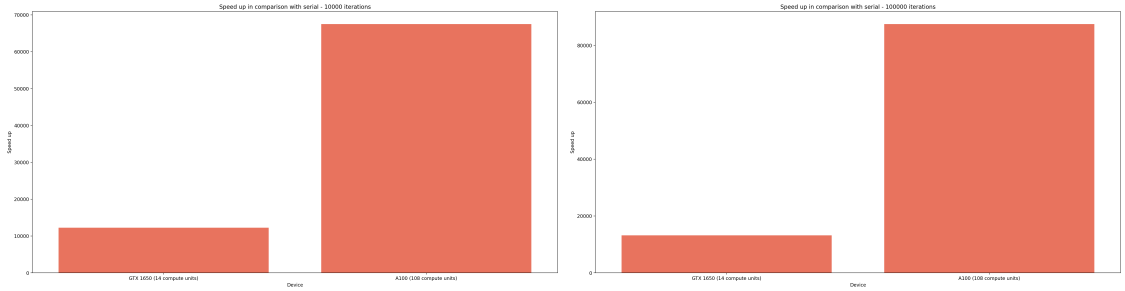
	Number of iterations	
Version	**10000**	**100000**
Serial	3398.30	33871.42
OpenMP - 10 threads	891.07	9454.07
GTX 1650	0.277460	2.566046
A100	0.050333	0.387145

Table 2. Execution time, in seconds, by number of threads and maximum number of iterations

Based on the execution times presented in Table 2, we have computed the speedup achieved by the parallel versions relative to the serial implementation and the OpenMP version. The speedup is defined in Equation 2, where T_s denotes the execution time of the serial version, T_p is the execution time of the parallel version, and p represents the number of processors used. In contrast with another parallel programming approach such as OpenMP, we are unable to controlling how many compute units should be or can be used by our program, for that reason, we can not ensure how many processors were used by our experiments. Ideally, we would want our implementation to use all available compute units in order to increase the number of kernels executed at the same time. To allow us to calculate the speedup we have used as the number of used processors the amount of available devices' compute units.

$$S(p) = \frac{T_s}{T_p} \tag{2}$$

In Figure 3, it is shown the speedup, in comparison with our serial version, for each device in terms of the number of iterations executed. The A100, on the right, yields a higher speedup in comparison with GTX 1650, on the left. This huge difference can be explained by the difference between number of cores available by each accelerator device. From 3.1.1 it is described the contrast between devices cores count.



(a) 10000 iterations **(b) 100000 iterations**

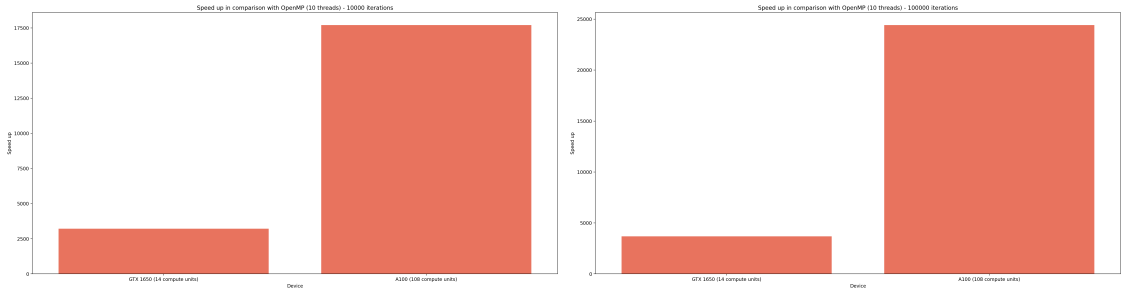
Figure 3. Speed up in comparison with serial version per device

When comparing the speedup with the parallel version using OpenMP with 10 threads, we see the same pattern in terms of behavior, the main difference is the scale, the speedup with OpenMP in four times less in comparison with the serial version. This result is shown in Figure 6.

In addition to evaluating the speedup achieved by our parallel executions, we also assessed the efficiency of each implementation under different configurations. Efficiency reflects how effectively computational resources are utilized to solve a given problem. It is defined by Equation 3, where $S(p)$ represents the speedup obtained with p processors, and p is the number of processors used, in our case, the number of compute units available.

$$E(p) = \frac{S(p)}{p} \tag{3}$$

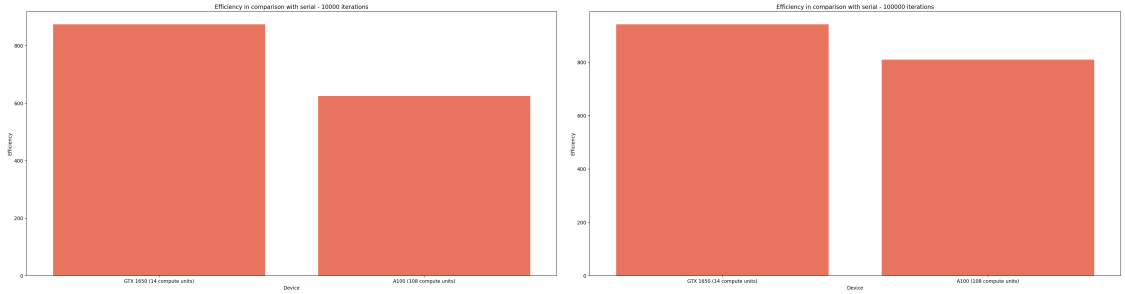
Figure 5 shows that efficiency is lower when using a device with more compute units. Although there was a difference in efficiency between devices, both devices



(a) 10000 iterations **(b) 100000 iterations**

Figure 4. Speed up in comparison with OpenMP version per device

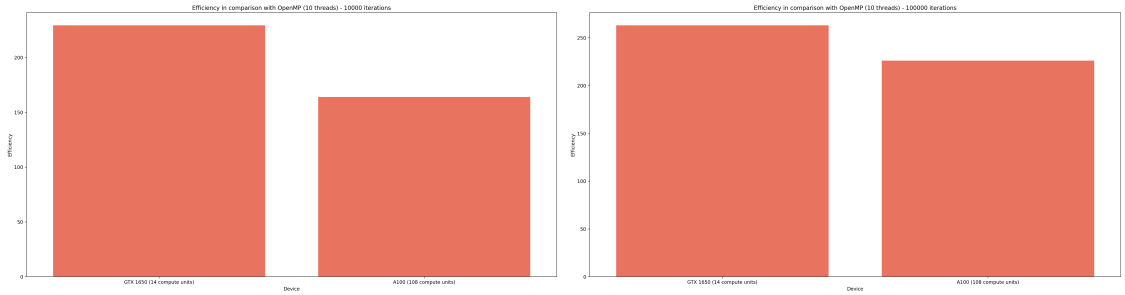
achieved a huge increase in efficiency of about 900 and 800 times for GTX 1650 (on the left) and A100 (on the right) respectively.



(a) 10000 iterations **(b) 100000 iterations**

Figure 5. Efficiency in comparison with serial version per device

The same pattern depicted in speedup is shown in Figure 6 for the efficiency. The only difference is that the efficiency is slower in comparison with the parallel version using OpenMP.



(a) 10000 iterations **(b) 100000 iterations**

Figure 6. Efficiency in comparison with OpenMP version per device

Figure 7 shows the result of a Mandelbrot generated using 100 iterations.

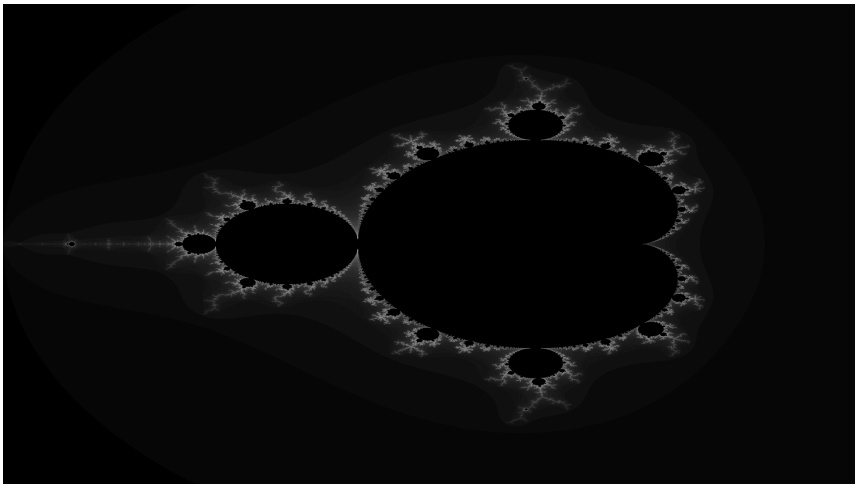


Figure 7. Mandelbrot result using 100 iterations

5. Conclusion

This report presented the development, execution, and analysis of a parallel implementation of the Mandelbrot set using the escape-time algorithm with a heterogeneous parallel programming environment. Our experiments were conducted in two different devices with different number of compute units, evaluating the impact of the maximum number of iterations allowed per pixel during the escape-time calculation in the execution time. Our findings indicate that the problem is well-suited for parallelization at the pixel level where each work-item is able of processing a piece of the problem independently. We have discovered that although this problem can take advantage of using more computational power to improve execution time, the problem is not efficient when using a device with too much processing power when comparing A100 and GTX 1650. Besides that, we were able to see, validate and analyze the advantages of employing a heterogeneous parallel programming model to solve a problem that can be executed in parallel to reduce the execution time and use resources available by hardware.

References

The Khronos Group (2020). *OpenCL 3.0 API Specification*. Khronos Group. `https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`.

Tompson, J. and Schlachter, K. (2012). An introduction to the opencl programming model. *Person Education*, 49:31.