

Implementation and Analysis of Fractals Shapes using GPU-CUDA Model

Amira Bibo Sallow

Dept. of Computer Science, Nawroz University, Dohuk, Kurdistan-region, Iraq

ABSTRACT

The rapid evolution of floating-point computing capacity and memory in recent years has resulted Graphics Processing Units (GPUs) an increasingly attractive platform to speed scientific applications and process large amount of data on time. Fractals have many implementations that involve faster computation and massive amounts of floating-point computation. In this paper, constructing the fractal image algorithm has been implemented both sequential and parallel versions using fractal Mandelbrot and Julia sets. Central Processing Unit (CPU) was used for the execution in sequential mode while GPU array and CUDA kernel was used for the parallel mode. The evaluation of the performance of the constructed algorithms for sequential structure using CPUs (2.20 GHz and 2.60 GHz) and parallelism structure for various models of GPU (GeForce GTX 1060 and GeForce GTX 1660 Ti) devices, calculated in terms of execution time and speedup to compare between CPU and GPU maximum ability. The results showed that the execution on GPU using GPU array or CUDA kernel is faster than its sequential implementation using CPU and its execution time increases exponentially while the execution time for GPU CUDA kernel model increases linearly. And the execution using the CUDA kernel is faster than the execution using GPU array, and the execution time between GPU devices was different, GPU with (Ti) series execute faster than the other models.

Keywords: Fractal, Fractal shapes, Mandelbrot & Julia set, self-similar, GPU, CUDA model, speedup.

1. Introduction

The Graphics Processing Units (GPUs) are an important development point due to the rapid advancement of the technology and in particular in video games and 3D applications. There is also a continuous increase in the development of GPUs for general computing activities. As multicore hardware is expanding rapidly, and to improve the computational performance in a various application, the reliance on the GPUs is quickly increased (Haji et al., 2020).

In several scientific research areas, the GPU achieves great success in high-performance computing, such as gene engineering, data mining computer games and entertainment software, medical sciences, engineering sciences, astronavigation, et al (Razian & MahvashMohammadi, 2017). Attaching more parallel computing resources makes GPU great compared to CPU in the floating-point calculation, particularly on large-scale data (Wang et al., 2015).

Escape-time fractals are a method for efficiently constructing fractal shapes at each point in space using an equation or recurrence relationship. Nowadays, with the development of computer hardware,

computer-generated models are becoming more and more real (Divya Udayan J, 2013). In certain cases, the algorithms for constructing various fractal shapes usually require large quantities of floating-point computation, which modern GPUs adapted for (Sallow & Abdullah, 2014).

In handling tasks with usual data access patterns, the GPU's parallel processing power can be completely utilized (Xiaodong Liu et al., 2014). Therefore, this paper explores the use of GPU to accelerate the construction of escape-time fractals (Mandelbrot set and Julia set). Matlab2020a is the programming language used for CPU and simple GPU implementations and CUDA programming model with Matlab2020a for using GPU compute kernel. The evaluation of the performance of the constructed algorithms for sequential structure using CPU and parallelism structure for various models of GPU devices, calculated in terms of execution time and speedup to compare between them.

The remaining section of this paper is structured as follows; in section 2 the literature review is given.

Section 3, "Fractals" describes fractals and their features, applications and common fractal shape sets. Section 4, "Graphics processing unites" defines the structure and types of GPU and its applications. Section 5, "organization of the implementation models" explains the implementation details of the algorithms in CPU and GPU. Section 6, "GPU hardware platform" describes the configuration of the used GPU devices. Section 7, illustrates the implementation results and analyze it. Section 8, explains the conclusions of the work.

2. Literature Review

(Zhang & Xu, 2011) simulated Mandelbrot set and Julia set via MATLAB and implemented complex calculations, large amounts of data storage and and output dependent on System-on-a-Programmable-Chip (SOPC) embedded in the Altera FPGA unit. It is revealed that at various iterations and complex parameters, but similar at different sizes, the fractal graphics appear differently. The study revealed that, based on a smaller hardware architecture, fractal algorithms and computer graphics can be implemented faster.

(Divya Udayan J, 2013) described the modeling of the fractal 2D model as the Mandelbrot and the fractal 3D model as terrain. To demonstrate the issue of natural objects with the use of classical geometry, they proposed a hardware-accelerated fractal-based rendering approach for natural environments. The comparison between the serial algorithm and a parallel algorithm was developed using the CUDA model using a GeForce GTX 650 GPU. The parallel speedup implemented was on average 2X times faster than its sequential implementation.

(Sallow & Abdullah, 2014) found that the construction of the fractal shapes using GPU Arrays in parallel modes was more powerful and faster than the construction using serial mode CPUs. Also, they found that in many cases, the algorithms for creating various fractal shapes usually require huge quantities of

floating-point computation for that reason, because of that they used GPU devices which are equipped for parallel computations with lots of arithmetic operations. The Sierpinski Gasket set was used as a fractal structure for their simulation.

(Anthony Atella, 2018) developed the Chaos application to fix the issue of online tools with restricted fractals, rendering techniques, and shaders. Which also struggle in a reusable way to abstract these concepts. This implies that it is important to learn several programs and interfaces and use them to fully research the topic. The Chaos, which is an abstract rendering program for fractal geometry.

The goal of Gilbert Gutabaga Hungilo (Hungilo et al., 2020) in 2020, was to explain the Mandelbrot Set construction simulation by comparing the results of the Python function generating the Mandelbrot set in three cases, using pure Python, Numba, and Numba CUDA running the Python function simultaneously. Using an iterated function system (IFS), the fractal is simulated based on the number of iterations and the size of the formed fractals. The simulation was accelerated using Numba that uses (Just In Time Python) code compilation, and Numba CUDA on GPU. Based on the performance results, the GPU simulation was quicker than the CPU models.

In the previous works, many Fractals construction methods were implemented using different devices. Some of them just used the simulation to construct many fractal shapes, while others compare implementation on CPU and GPU. In this paper, the comparison was done using three different models one on CPU and two on GPU. Also, it provides a very well distinction between GPU devices that other developers and researchers may employ.

3. Fractals

Fractals are natural or artificial structures or geometric patterns that provide a high degree of scale invariance or self-similarity. Fractal-related scale invariance means that the scale is not significant to the outcomes.

It is possible to observe structures that are invariant in scale from various distances, but the object still looks the same. Self-similar systems have parts that are identical to themselves. It is also clear that inside the object the entire structure can be identified. Because of these two properties, very complex structures, when recognized as fractals, can be based on simple laws. There are fractals nearly everywhere in the nature. But it is often not easy to determine objects to be fractal objects. But the rules of fractals help to understand them better and calculate or simulate them. There are a lot of objects in the nature has the fractals properties (mountains, blood vessels, snowflakes, etc.) (Mandelbrot, 2004). Fig (1) shows some fractal plants.



Fig.1. Fractals plants

simply iterating the function again and again, a fractal can be generated. For instance, if a function is $f(y)$ and the initial value is $y = x$, then $f(x)$, $f(f(x))$, $f(f(f(x)))$, etc. That would have been the outcome. It could be possible that after each iteration it will give greater and greater value by iterating function (Negi et al., 2014).

3.1 Properties of Fractal

A fractal has the following Combined features (Belma & Sonay, 2016):

- The parts are of the same shape or structure as the whole unless they are of different size and maybe slightly deformed;
- Its shape, and remains are extremely irregular or fragmented irrespective of the size of the examination;

- It includes 'different components,' the sizes of which are very varied and cover a wide range;
- Composition according to iteration;
- Scale-independent

3.2 Fractals applications

Fractals are the distinctive, irregular patterns left behind by the chaotic world's unexpected movements at work. Most applications of fractals below are: (Biswas et al., 2018)

- Astronomy: More evidence on the distribution of matter in the universe is required by cosmologists to prove (or not) that the universe is a fractal.
- Nature: Modeling nature visually (coastlines, mountains, soil erosion and to analyze seismic patterns).
- Computer science: the most beneficial is image compression.
- Fluid mechanics: allows engineers and physicists understand complex flows more effectively (Turbulent flows).
- Telecommunications: reducing the antenna size and weight (Fractenna company).
- Surface physics: explain the roughness of surfaces.
- Medicine: biosensor interactions can be studied.

3.3 Fractals Sets

3.3.1 Sierpinski triangle

The Sierpiński triangle also referred to as the Sierpiński gasket or Sierpiński sieve, is an elegant static fractal set of an equilateral triangle's exact shape, iteratively split into smaller equilateral triangles. it can be constructed by continuously performing a method of joining the midpoints of each side of the triangle to form four different triangles and slicing out the middle triangle (Sawant, n.d.), as shown in Fig(2).

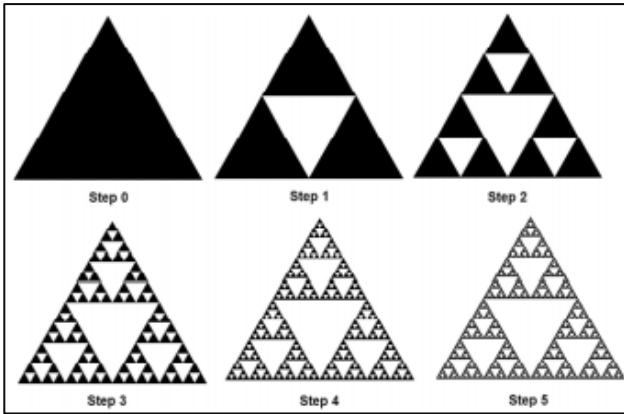


Fig. 2. Sierpinski Triangle

3.3.2 Mandelbrot set

In mathematics, the Mandelbrot set is an instance of a fractal set of complex numbers as well. It is critical for chaos theory and can be represented with the equation (1): (Negi et al., 2014)

$$z_{n+1} = z_n^2 + c \dots \dots (1)$$

Where complicated numbers are c and z and n is 0 or a positive integer (natural number). For Z , the starting value is always (0.0). C is the constant component that specifies the position in the complex plane of the iteration sequence. The set of Mandelbrot is now the set of (C)s whose outcomes are not diverging to infinity but remain under those limits. Fig (3) shows the shape of Mandelbrot set.

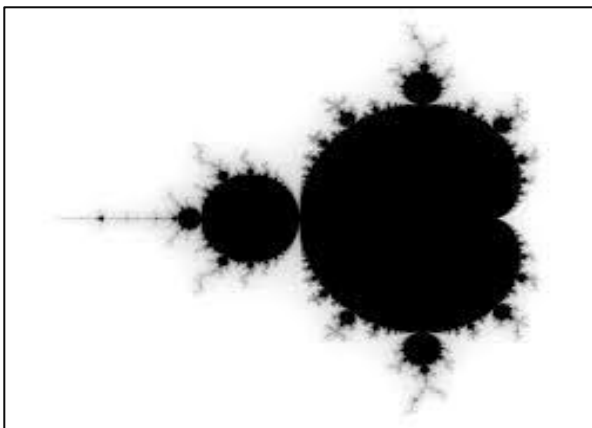


Fig.3. Mandelbrot set

3.3.3 Julia Set

A particular Julia set can be described by a point matching its constant c value in the Mandelbrot set and the look of an entire Julia set is generally identical in style to that of the Mandelbrot set at that spot. Usually, points near the edges of the Mandelbrot set give the

most fascinating Julia sets. In the Mandelbrot set, c varies with each pixel and equals $(x + yi)$, where x and y are the image dimensions. All Julia sets are a map of the Mandelbrot set, since it requires a different c at each position, like it is transformed through space between one Julia set to the other (Mandelbrot, 1982).

Fig (4) shows one of Julia set shapes.

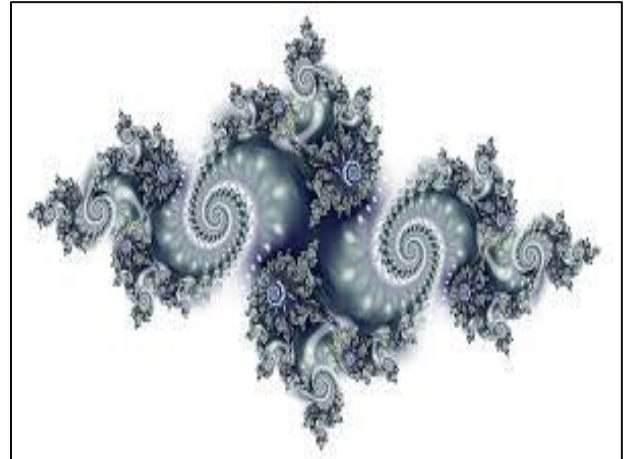


Fig.4. Julia set

4. Graphics Processing Units

A graphics processing unit (GPU) is a specialized electronic circuit designed to automatically access the memory in order to speed up the production of images in a frame buffer designed for display system output. In embedded systems, cell phones, personal devices, workstations, and game consoles, GPUs are included (Kirk & Hwu, 2013). The GPU is extremely well-suited to resolve concerns that can be presented as data-parallel computations. The same program is run on several data elements in parallel with the high ratio of arithmetic operations to memory operations. Their intensely parallel nature makes them more effective for algorithms that process large amounts of data in parallel than general-purpose central processing units (CPUs) (Jimenez et al., 2017).

In GPUs, many more transistors act like they can process data arrays rather than flow control of many sequential processing threads. Fig (5) shows how much area different circuits in CPUs and GPUs occupy: (Nogues et al., 2020)

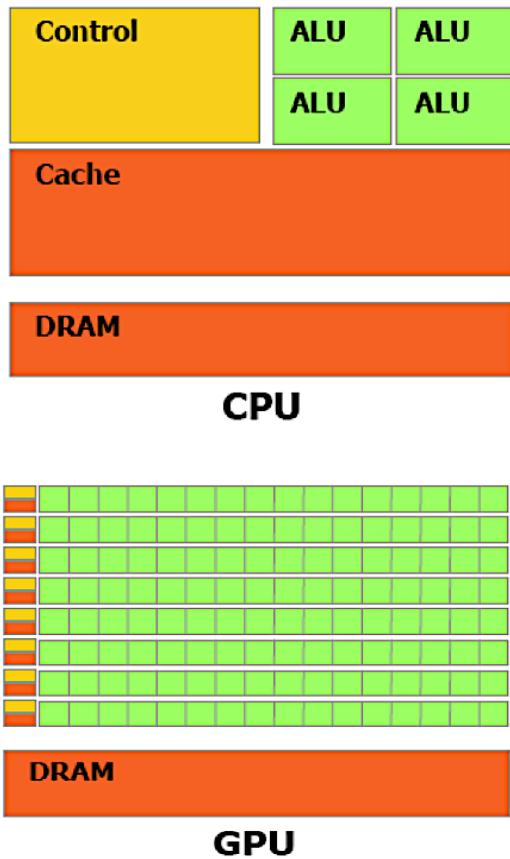


Fig.5. CPUs vs GPUs

4.2 GPUs Types (Kirk & Hwu, 2013)

- Dedicated graphics cards: the most powerful class uses an expansion slot to connect with the motherboard.
- Integrated graphics processing (IGP): can be done as a part of the (northbridge) chipset into the motherboard, or with the Processor on the same die.
- Processing hybrid graphics: share memory with the system and have a relatively small dedicated memory buffer to help make up for the high latency of the RAM.
- External GPU (eGPU): similar to a large external hard drive located outside the computer's housing.
- General-purpose computing on graphics processing units: Used as a modified version of a stream processor that runs compute kernels. This principle converts the huge computational power of the modern graphics accelerator into general

computational power to be used for various forms of parallel operations.

4.3 GPU applications

Some of the fields where GPUs are used for general purpose computing are as follows: (Kirk & Hwu, 2017)

- Signal, image and video processing, computer vision.
- Artificial intelligence, database processes.
- Weather forecasting, geoprocessing, climate analysis.
- Molecular modeling, astrophysics, quantum mechanical science.
- computational finance.
- Intrusion detection, cryptography, and cryptanalysis.
- Automation of electronic design.
- Increasing computational resources for distributed projects.

4.4 Compute Unified Device Architecture (CUDA) kernel

CUDA is a model developed by Nvidia for the parallel computing and application programming interface (API). A CUDA-enabled GPU can be used by software developers and software engineers for general purpose processing, an approach called General-Purpose computing on Graphics Processing Units (GPGPU). The CUDA platform is a software layer that provides direct access to the simulated instruction set of the GPU and parallel processing elements for computer kernel execution. A kernel is a procedure compiled independently from but used by a running main program on CPU for high throughput accelerators (such as Digital Signal Processors (DSPs) or Field-Programmable Gate Arrays (FPGAs), GPUs). Sometimes they can be called compute shaders, sharing execution units on GPUs with vertex shaders and pixel shaders, but they are not restricted to execution on one system class or graphics APIs (Liu et

al., 2019). They can be classified through separate programming languages such as Open Computing Language (OpenCL C), or (compute shaders) which are written in a shading language, or are directly implemented in application code written in a high-level language, as C++AMP (Kirk & Hwu, 2017).

5. Organization of The Implementation Models

Mandelbrot set and Julia set were used as a case study to be constructed using GPU devices. The goal of using these devices is to greatly reducing the time required to construct fractals shapes and any other shapes that required large amounts of mathematic computations. Three models had used in this paper to construct fractal shapes. The first is the CPU sequential model, and the second is the GPU parallel model using GPU array, while the third one uses the GPU parallel CUDA model. All the algorithms have been executed using MATLAB 2020a.

5.1 Sequential CPU Model

Fractal shapes constructed sequentially using a single threaded approach, and all calculations are performed on the host CPU by MATLAB. Table (1) shows the configurations of CPUs that used in the implementation.

Table (1): Complete CPU configurations

| System | Processors Name | Speed | RAM | Cores Numbers | System type |
|-------------------------|-----------------|----------|-------|---------------|-------------|
| Cp1_GeForce GTX 1060 | Intel-i7-8750H | 2.20 GHz | 16 GB | 6 | 64-bit |
| Cp2_GeForce GTX 1660 Ti | Intel-i7-9750H | 2.60 GHz | 32 GB | 6 | 64-bit |

Fig (6) shows the steps to construct Mandelbrot set and Julia set fractals shapes on the host CPU.

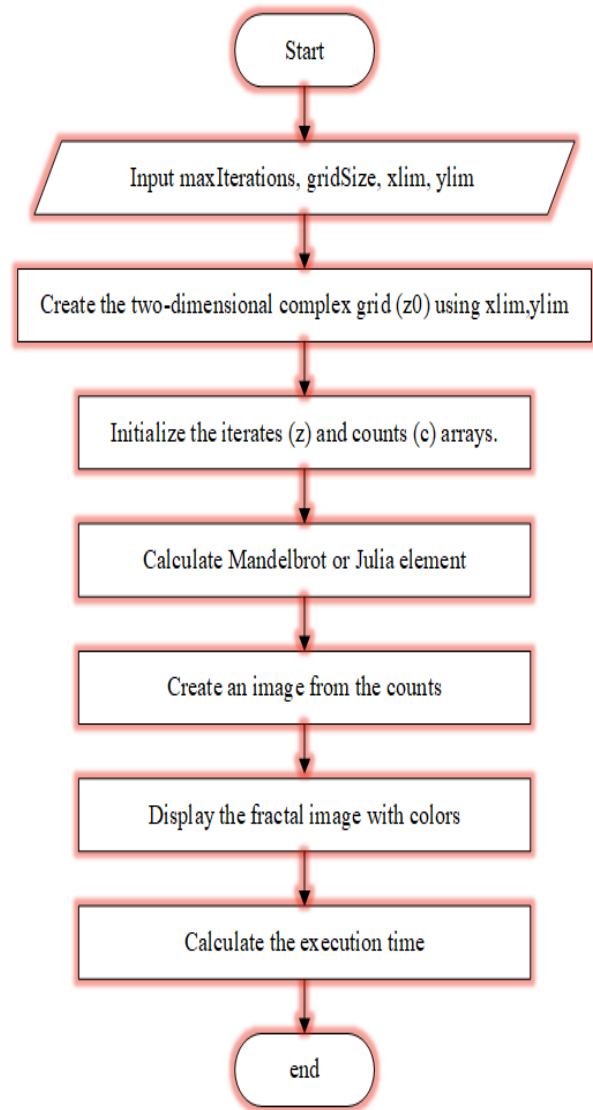


Fig. 6. CPU model execution flowchart

5.2 GPU Simple Model (using gpuArray):

The same algorithm has used to construct fractal shapes, but the input coordinates had moved to be stored on the GPU using gpuArray. This technique allows MATLAB to work with the same data array on the GPU using the same code. Also, this technique provides some speedup with no coding cost by using gpuArray method in MATLAB to convert the input data to be storied on the GPU. Fig (7) shows the steps to construct Mandelbrot set and Julia set fractals shapes on the GPU using gpuArray method.

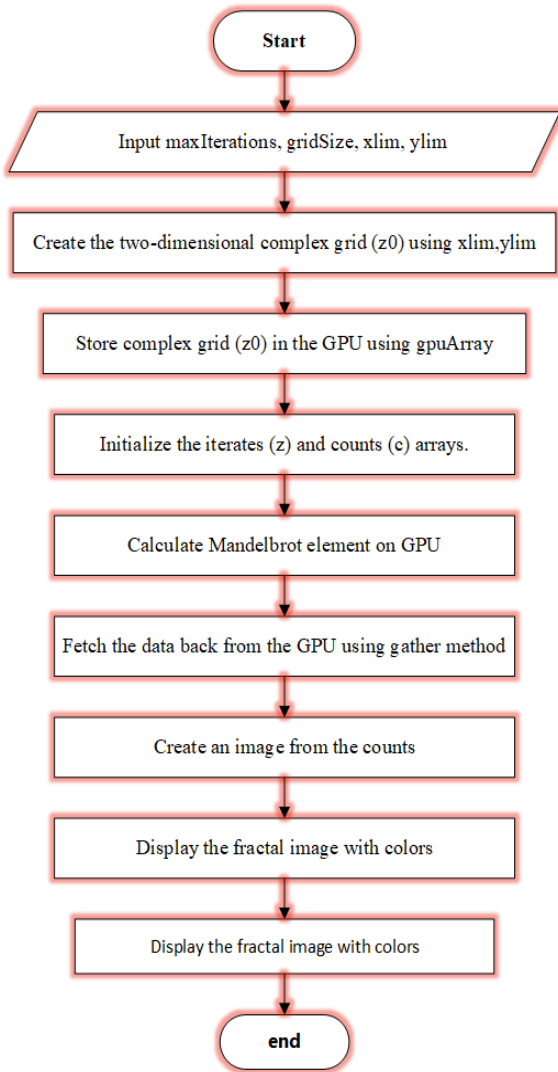


Fig.7. GPU Simple model execution flowchart

5.3 GPU CUDAKernel Model

The CUDA model constructs the fractal shape differently. This model uses a kernel in which each thread computes the value of each pixel, and then colors it. Fig (8) shows the CUDA kernel code for constructing Mandelbrot fractal shape, saved in Mandelbrot.cu file written in C programming language. Fig (9) shows the steps to construct the Mandelbrot set and Julia set fractals shapes on the GPU using the CUDA model. The CUDAKernel method is used to call the kernel in MATLAB, requiring the user to define the thread and block configurations to use. To construct the Julia set the same kernel used but the values of Cs changed.

```

/** The actual Mandelbrot algorithm for a single location */
_device_ double dolterations( double const realPart0, double const imagPart0, double const escapeRadius, unsigned int const maxIters)
{
    // Initialise: z = z0
    double const escapeRadius2 = escapeRadius*escapeRadius;
    double realPart = realPart0;
    double imagPart = imagPart0;
    unsigned int count = 0;
    // Loop until escape
    while ( ( count <= maxIters ) && ((realPart*realPart + imagPart*imagPart) <= escapeRadius2) ) {
        ++count;
        // Update: z = z^2 + z0;
        double const oldRealPart = realPart;
        realPart = realPart*realPart - imagPart*imagPart + realPart0;
        imagPart = 2.0*oldRealPart*imagPart + imagPart0;
    }
    // Correct final position for smooth shading
    double const absZ2 = ( realPart*realPart + imagPart*imagPart );
    if (absZ2 < escapeRadius2)
    {
        return double(count) + 1.0 - log( log( escapeRadius2 ) / 2.0 ) / log(2.0);
    }
    else {
        return double(count) + 1.0 - log( log( absZ2 ) / 2.0 ) / log(2.0);
    }
}
    
```

Fig.8. CUDA kernel code for constructing fractal shape.

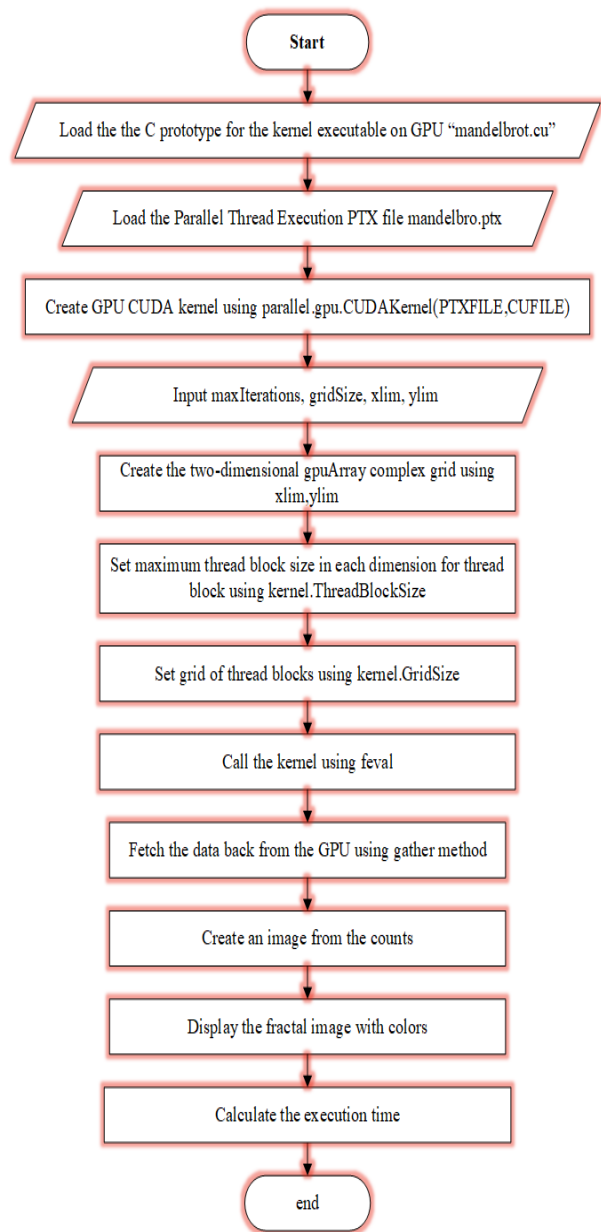


Fig.9. GPU CUDAKernel model execution flowchart

6. GPU Hardware platform

A thread is generated within each pixel in the GPU architecture and a specific thread id is assigned to each thread. In parallel, each thread executes the algorithm and outputs the results. NVIDIA hardware is used for this implementation. Two laptop computers were used each one supported with GPU hardware and their complete specification as shown in fig (10.a,10. b).

```

CUDADevice with properties:
    Name: 'GeForce GTX 1060'
    Index: 1
    ComputeCapability: '6.1'
    SupportsDouble: 1
    DriverVersion: 9.2000
    ToolkitVersion: 9.1000
    MaxThreadsPerBlock: 1024
    MaxShmemPerBlock: 49152
    MaxThreadBlockSize: [1024 1024 64]
    MaxGridSize: [2.1475e+09 65535 65535]
    SIMDWidth: 32
    TotalMemory: 6.4425e+09
    AvailableMemory: 5.2463e+09
    MultiprocessorCount: 10
    ClockRateKHz: 1733000
    ComputeMode: 'Default'
    GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
    CanMapHostMemory: 1
    DeviceSupported: 1
    DeviceSelected: 1
    
```

a) Computer 1: GeForce GTX 1060

```

CUDADevice with properties:
    Name: 'GeForce GTX 1660 Ti'
    Index: 1
    ComputeCapability: '7.5'
    SupportsDouble: 1
    DriverVersion: 11.1000
    ToolkitVersion: 9.1000
    MaxThreadsPerBlock: 1024
    MaxShmemPerBlock: 49152
    MaxThreadBlockSize: [1024 1024 64]
    MaxGridSize: [2.1475e+09 65535 65535]
    SIMDWidth: 32
    TotalMemory: 6.4425e+09
    AvailableMemory: 5.0335e+09
    MultiprocessorCount: 24
    ClockRateKHz: 1590000
    ComputeMode: 'Default'
    GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
    CanMapHostMemory: 1
    DeviceSupported: 1
    DeviceSelected: 1
    
```

b) Computer 2: GeForce GTX 1660 Ti

Fig.10. MSI PC's NVIDIA hardware supported specification

The experimental uses single-precision operation on CPU and GPU implementation to construct the fractals shapes. Speedup fold (x) measures the acceleration of the algorithm in GPU implementation and is the time that the algorithm takes to run on the host CPU, divided by the time that takes to run on the GPU device. Table (2) reports the processing time taken by the fractal algorithm for three different iterations values (1000,5000,10000) for each fractal set (Mandelbrot, Julia). The plot between execution time for implementation fractal algorithm is shown in Fig (11), Fig (12), and Fig (13)

Table (2): Processing time of CPU and GPU systems.

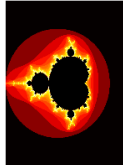
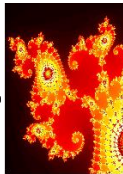
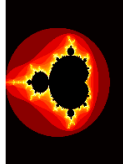
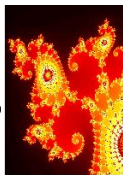
| Device | Fractal shape | Iterations | CPU (execution time) | GPU Array (execution time) | Speed up1 | GPU_CUDA Kernel (execution time) | Speed up2 |
|--------------------------------|---|------------|----------------------|----------------------------|-----------|----------------------------------|-----------|
| Computer1: GeForce GTX 1060 |  | 1000 | 7.63secs | 1.99secs | 3.8x | 0.05 secs | 171.4x |
| | | 5000 | 35.26secs | 10.77secs | 3.3x | 0.39 secs | 101x |
| | | 10000 | 71.60secs | 19.40secs | 3.7x | 0.47secs | 173x |
| |  | 1000 | 8.25secs | 2.01secs | 4.1x | 0.06secs | 157.5x |
| | | 5000 | 40.83secs | 10.43secs | 3.9x | 0.25secs | 167.2x |
| | | 10000 | 77.50secs | 21.33secs | 3.6x | 0.23secs | 373.1x |
| Computer2: GeForce GTX 1660 Ti |  | 1000 | 6.30secs | 0.94secs | 6.7x | 0.044 sec | 143x |
| | | 5000 | 28.32secs | 4.50secs | 6.3x | 0.095 secs | 298x |
| | | 10000 | 55.84secs | 8.93secs | 6.3x | 0.161 sec | 347x |
| |  | 1000 | 6.63secs | 1.52secs | 4.4x | 0.054 secs | 123x |
| | | 5000 | 30.17secs | 4.69secs | 6.4x | 0.055 secs | 548x |
| | | 10000 | 59.23secs | 9.39secs | 6.3x | 0.052 secs | 1139x |

Fig (11) shows the execution time on laptop 1 (Pc1) that supported with CPU (2.20 GHz) is slower that the execution time on labtop2 (Pc2) that supported with CPU (2.60 GHz)

7. Implementation Results

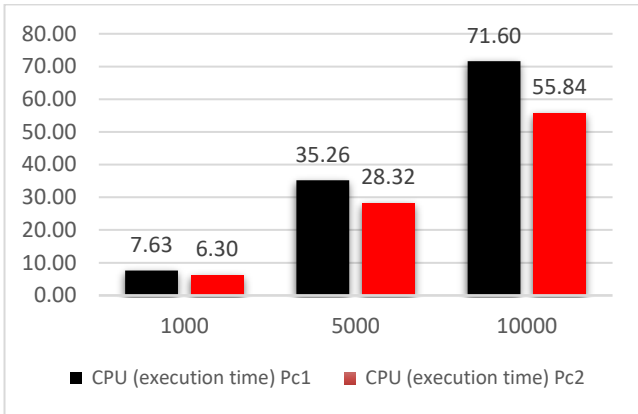


Fig.11. Comparison of cp1 and cp2 computation time on CPU

Fig (12) shows the execution time using GPU array on laptop 1 (Pc1) that supported with GPU (GeForce GTX 1060) is lower that the execution time on labtop2 (Pc2) that supported with GPU (GeForce GTX 1660 Ti).

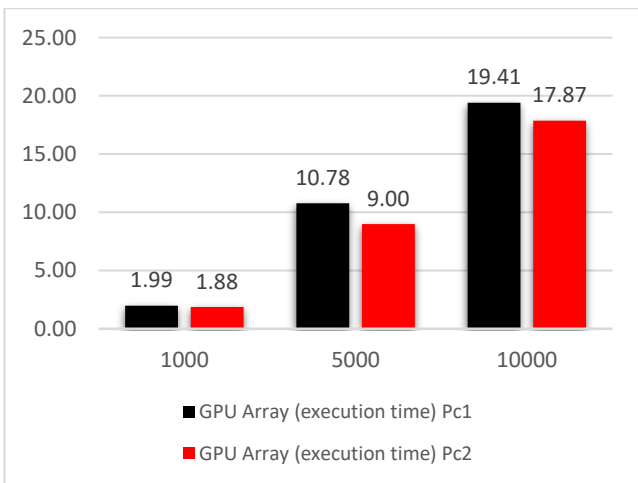


Fig.12. Comparison of cp1 and cp2 computation time on GPU Array

Fig (12) shows the execution time using CUDA kernel on laptop 1 (Pc1) that supported with GPU (GeForce GTX 1060) is lower that the execution time on labtop2 (Pc2) that supported with GPU (GeForce GTX 1660 Ti).

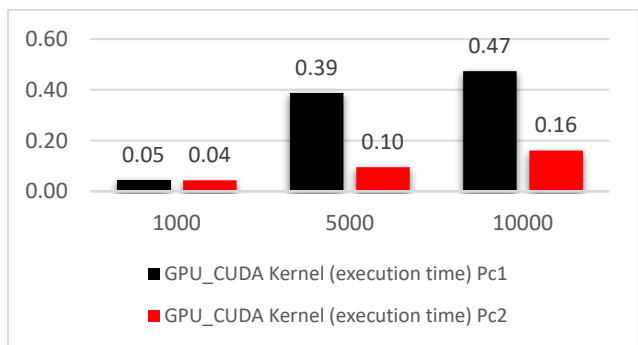


Fig.13. Comparison of cp1 and cp2 computation time on GPU CUDA

From the table (2) and Fig (11), Fig (12), and Fig (13)

graphs, the time needed by the GPU is almost constant, and as the image size increases, the CPU time increases exponentially, while the execution time for GPU CUDA kernel model increases linearly as showing table (3) and Fig (14).

Table (3): Generation Times for Vary Square Images

| | CPU | GPU Array | GPU_CUDA |
|--------|--------|-----------|------------|
| size | Pc1 | Pc1 | Kernel Pc1 |
| 10 | 0.16 | 0.14 | 0.03 |
| 100 | 0.86 | 0.16 | 0.04 |
| 1000 | 9.28 | 2.00 | 0.06 |
| 10000 | 84.30 | 21.07 | 0.23 |
| 100000 | 818.36 | 203.32 | 0.38 |

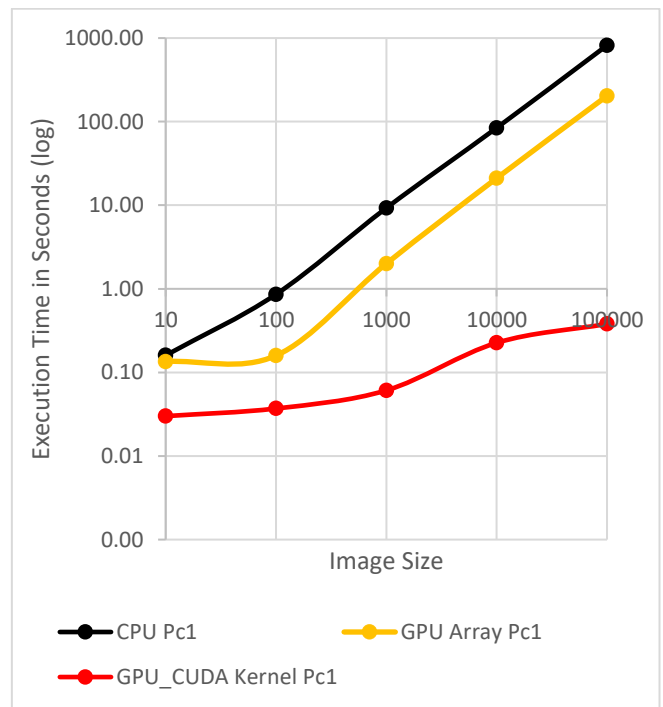


Fig.14. Comparison of Generation Times

8. Conclusion

The construction of fractals Mandelbrot and Julia sets were done in both serial and parallel modes using different hardwares supported with accelerated techniques to increase the speed of computation in fractal shapes. The evaluation of the performance of the constructed algorithms for sequential structure using CPUs (2.20 GHz and 2.60 GHz) and parallelism structure for various models of GPU (GeForce GTX

1060 and GeForce GTX 1660 Ti) devices, calculated in terms of execution time and speedup to compare CPU and GPU maximum ability. The results showed that execution on GPU using GPU array or GUDA kernel is faster than its sequential implementation. And the execution using the GUDA kernel is faster than the execution using GPU array. The execution time between GPU devices was different, GPU with (Ti: "Titanium") series execute faster than the other models and more powerful.

9. References

1. Anthony Atella. (2018). Rendering Hypercomplex Fractals. *Honors Projects Overview*, 44.
2. Belma, A., & Sonay, A. (2016). *Fractals and Fractal Design in Architecture*. 17(3), 10.
3. Biswas, H. R., Hasan, M., & Bala, S. K. (2018). *CHAOS THEORY AND ITS APPLICATIONS IN OUR REAL LIFE*. 19.
4. Divya Udayan J. (2013). Fractal Based Method on Hardware Acceleration for Natural Environments. *Future Technology Research Association International*, 4(3).
5. Haji, L. M., Zebari, R. R., Zeebaree, S. R. M., Mustafa, W., Shukur, H. M., & Ahmed, O. M. (2020). *GPUs Impact on Parallel Shared Memory Systems Performance*. 24(08), 9.
6. Hungilo, G. G., Emmanuel, G., & Pranowo. (2020). *Performance comparison in simulation of Mandelbrot set fractals using Numba*. 030007. <https://doi.org/10.1063/5.0000636>
7. Jimenez, L. I., Sanchez, S., Martan, G., Plaza, J., & Plaza, A. J. (2017). Parallel Implementation of Spatial-Spectral Endmember Extraction on Graphic Processing Units. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(4), 1247-1255. <https://doi.org/10.1109/JSTARS.2016.2645718>
8. Kirk, D., & Hwu, W. (2017). *Programming massively parallel processors: A hands-on approach* (Third edition). Elsevier.
9. Kirk, D., & Hwu, W. W. (2013). *Programming massively parallel processors: A hands-on approach* (2. ed). Elsevier, Morgan Kaufmann.
10. Liu, Y., Cui, H., & Zhao, R. (2019). Fast Acquisition of Spread Spectrum Signals Using Multiple GPUs. *IEEE Transactions on Aerospace and Electronic Systems*, 55(6), 3117-3125. <https://doi.org/10.1109/TAES.2019.2902695>
11. Mandelbrot, B. B. (1982). *The fractal geometry of nature*. W.H. Freeman.
12. Mandelbrot, B. B. (2004). *Fractals and Chaos*. Springer New York. <https://doi.org/10.1007/978-1-4757-4017-2>
13. Negi, A., Garg, A., & Agrawal, A. (2014). A Review on Natural Phenomenon of Fractal Geometry. *International Journal of Computer Applications*, 86(4), 1-7. <https://doi.org/10.5120/14970-3157>
14. Nogues, O. C. i, Pascual, D., Onrubia, R., & Camps, A. (2020). Advanced GNSS-R Signals Processing With GPUs. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13, 1158-1163. <https://doi.org/10.1109/JSTARS.2020.2975109>
15. Razian, S. A., & Mahvash Mohammadi, H. (2017). Optimizing Raytracing Algorithm Using CUDA. *Italian Journal of Science & Engineering*, 1(3), 167-178. <https://doi.org/10.28991/ijse-01119>
16. Sallow, A., & Abdullah, D. (2014). Constructing Sierpinski Gasket Using GPUs Arrays. *International Journal of Computer Science Issues*, 11(6), 3.
17. Sawant, V. G. (n.d.). *DESIGN OF HIGH GAIN FRACTAL ANTENNA*. 6(1), 8.
18. Wang, G., Zomaya, A., Martinez, G., & Li, K. (Eds.). (2015). *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part I* (Vol. 9528). Springer International Publishing. <https://doi.org/10.1007/978-3-319-27119-4>
19. Xiaodong Liu, Mo Li, Shanshan Li, Shaoliang Peng, Xiangke Liao, & Xiaopei Lu. (2014). IMGPU: GPU-Accelerated Influence Maximization in Large-Scale Social Networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(1), 136-145. <https://doi.org/10.1109/TPDS.2013.41>
20. Zhang, X., & Xu, Z. (2011). Implementation of Mandelbrot set and Julia Set on SOPC platform. *2011 International Conference on Electronics, Communications and Control (ICECC)*, 1494-1498. <https://doi.org/10.1109/ICECC.2011.6066355>