HIGH PERFORMANCE COMPUTING - Mandelbrot

Michele Frattini 4878744 - Luigi Timossi 4819664

Contents

1 Introduction

The following report describes the various processes aimed at optimizing the program for the calculation and generation of the Mandelbrot fractal using OpenMP and CUDA. Subsequently, we reported and analyzed the obtained results.

1.1 The Mandelbrot Fractal

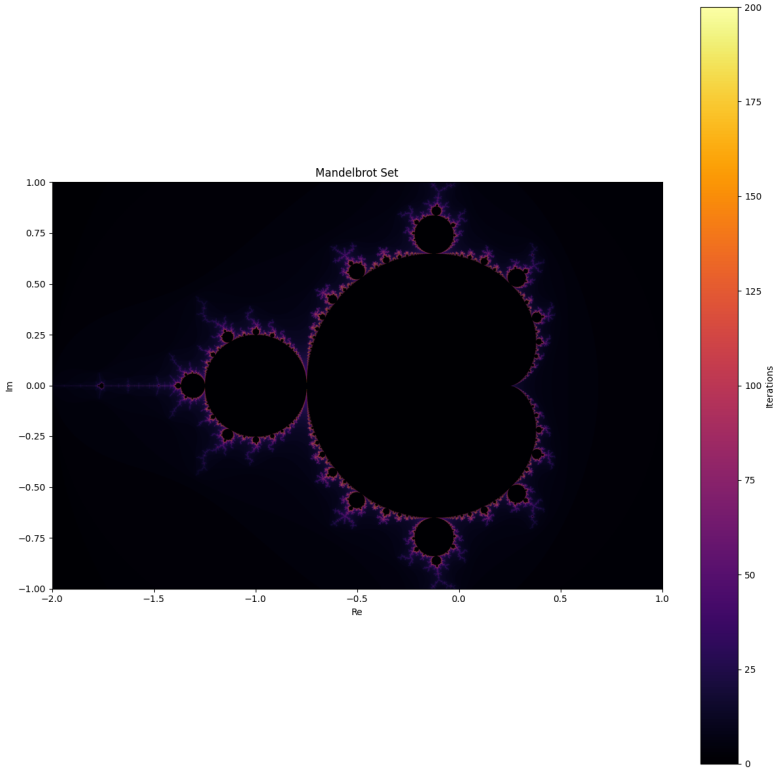

Figure 1: Mandelbrot with 200 Max iteration for better visualization

The Mandelbrot set is a well-known fractal that emerges by iterating a simple mathematical function on the complex plane. It is defined by the equa-

tion:

$$Z_{n+1} = Z_n^2 + C \tag{1}$$

where Z and C are complex numbers, and the iteration starts with $Z_0 = 0$. The Mandelbrot set consists of all points C for which the sequence does not diverge to infinity.

Since the relation is quadratic, the exit condition is based on the modulus of Z_n: if $|Z_n| > 2$, then the point C does not belong to the set. In fact, for high values of $|Z_n|$, the square term dominates, and the modulus tends to infinity.

Generating the Mandelbrot set is a computationally demanding process since it is necessary to iterate the function many times for each point in the grid of the complex plane. Increasing the resolution and the maximum number of iterations required to achieve finer details significantly raises the computational complexity.

2 Machine specifications

For the final project, we used two types of machines: the computers in the SW2 lab of the department and the virtual machine provided by Google Colab.

2.1 SW2

To find all the characteristics of the SW2 machine processor, we used the command `cat /proc/cpuinfo`:

- 20 Intel i7-12700 processors
- 12 physical cores
- 8 P-cores with Hyper-Threading (16 threads)
- 4 E-cores without Hyper-Threading (4 threads)
- AVX architecture

Hyper-Threading is an Intel-exclusive technology where a physical core acts as two virtual logical cores, allowing a single processor to execute multiple threads concurrently.

2.2 Colab

To find the specifications of the Colab virtual machine, we used the commands `!nvcc /content/devicequery.cu -run` and `nvidia-smi`, obtaining the following:

- NVIDIA T400 Tesla

- Warp size 32
- Maximum threads per block of 1024
- Maximum block dimension of 1024, 1024, 64
- Maximum grid size of $2147483647 \times 65535 \times 65535$
- Approximately 16GB of GPU memory

3 Hotspot identification

To identify hotspots, we used the Intel Advisor GUI analysis tool, available on the department lab computers, compiling with the following command:

`icpx -g mandelbrot.cpp`.

The compilation flag `-g` tells the compiler to generate a level of debugging information in the object file.

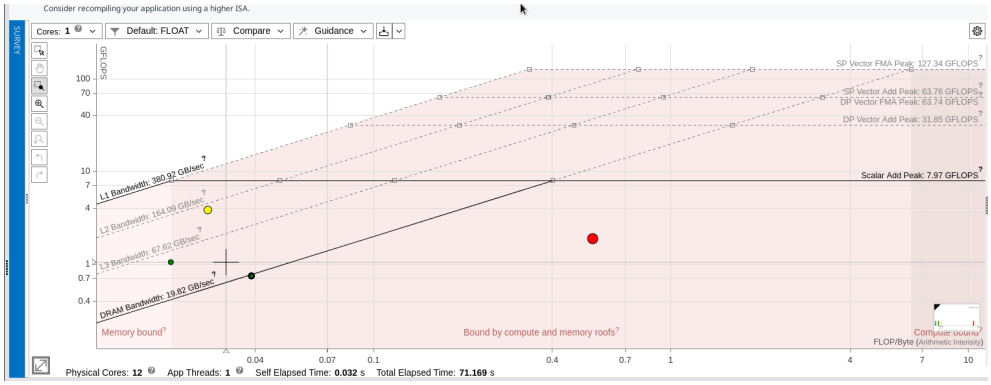

Figure 2: First screen of Intel Advisor GUI

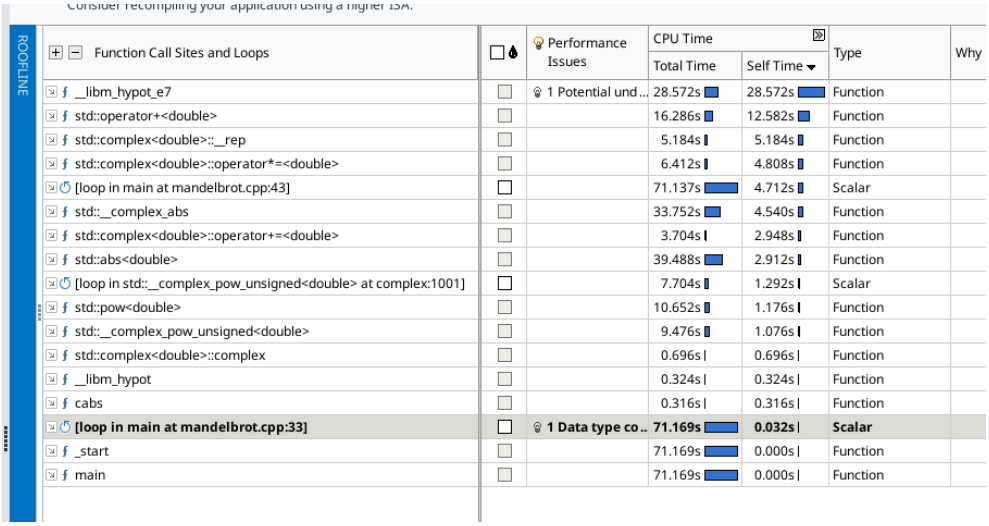

Figure 3: Second screen of Intel Advisor GUI

From the two images above, we can see that the main hotspots are:

- Mathematical operations of `_libm_hypot`, which performs the Euclidean distance
- Overloading of the `+` operator for double in the `std` library
- Conversion to `complex` from `double`
- `Power, addition, and absolute value` operations on complex numbers

Thus, operations and comparisons, when working with complex numbers, involve conversions and various internal functions to be completed. These operations occupy most of the execution time as they are computationally expensive.

Name	**Time in sec**	**Time/Total Time**
Internal loop:43	71.13	99.96%
_limb_hypot_e7	28.57	40.14%
+ <**double**>	16.28	22.88%
abs<double>	39.48	55.49%
pow <**double**>	10.65	14.97%
Total(_start)	71.16	100%

Table 1: Different time usages

The operation that takes the longest to execute is the abs for complex numbers, as it takes up more than half of the program's total execution time. The second

most expensive operation is the calculation of the Euclidean distance, indicating that most of the time spent is due to the use of complex numbers in the algorithm.

4 Vectorization issues

We used `icpx` to generate a vectorization report with the following command `icpx -g -qopt-report=3 mandelbrot.cpp`. The command generated two different files: mandelbrot.opt.yaml and mandelbrot.optrpt. In the latter, we found some vectorization issues, as shown in the image below 4.

```
Global optimization report for : main

LOOP BEGIN at mandelbrot.cpp (33, 5)
    remark #25408: memset generated
    remark #15553: loop was not vectorized: outer loop is not an auto-vectorization candidate.

    LOOP BEGIN at mandelbrot.cpp (43, 9)
        remark #25530: Stmt at line 0 sinked after loop using last value computation
        remark #15344: Loop was not vectorized: vector dependence prevents vectorization
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #15346: vector dependence: assumed FLOW dependence between z __t __x (1336:13) and z __t __x (1353:13)
        remark #25438: Loop unrolled without remainder by 2
    LOOP END
LOOP END
```

Figure 4: Vectorization issue

- The outer loop, according to the report, is not a valid candidate for vectorization, so it is not vectorizable
- The inner loop, however, cannot be vectorized because the value of z, for a given iteration i, depends on the previous iteration, resulting in a flow dependency
- Also within the inner loop, there is a `break` statement since, for different `pos` values, it is possible to obtain a `z` value that breaks the loop before reaching the maximum defined iteration. It is possible to remove it by replacing it with a ternary operator to attempt to force vectorization. However, this solution results in a high number of unnecessary operations and does not eliminate the dependency of the loop on previous iteration values

5 Configuration used

For our analysis we decided to increase the computational power by increasing the number of iterations to 5000 and the resolution to 5000.

We decided to consider several compiler configurations for our analysis. Each configuration uses a different set of flags.

5.1 Fast configuration

For the first configuration, we decided to consider only the `-fast` flag, obtaining the command:

```
icpx -fast -xHost mandelbrot.cpp
```

- **-fast** → According to the official guide intel, on a Linux IA-32 architecture and Intel 64 system, this flag is equivalent to: `-ipo, -O3, -no-prec-div, -static, and -xHost`
- **-ipo** → When specified, the compiler performs inline function expansion for calls to functions defined in separate files
- **-O3** → Enables maximum optimization
- **-no-prec-div** → It enables optimizations that give slightly less precise results
- **-static** → This option prevents linking with shared libraries. It causes the executable to link all libraries statically
- **-xHost** → Optimizes for the Intel host CPU, the compiler will automatically turn on the code-generation flags corresponding to the highest instruction set supported by the CPU

Since the compiler was giving us a warning (icpx: warning: -x HOST after last input file has no effect [-Wunused-command-line-argument]), we decided to explicitly add the -XHost flag as well. This is probably because the -fast flag appends its options at the end of the command.

5.2 Fast and ffast-math configuration

In this configuration, we added the flag to optimize mathematical operations, with the following command:

```
icpx -fast -xHost -ffast-math mandelbrot.cpp
```

- **-ffast-math** → Enables fast math calculations (may affect precision)

5.3 Classic best configuration

In the two previous laboratories, we found this configuration to be the best:

```
icpx -fast -xHost -ffast-math -fp-model fast mandelbrot.cpp
```

- **-fp-model fast** → This option controls the semantics of floating-point calculations, and the fast keyword allows aggressive floating-point optimizations

6 Configuration time analysis and comparison

After choosing the following configurations, we decided to conduct a study on the execution time of each. Each configuration is used to compile the `mandelbrot.cpp` code, and each executable is run 10 times, saving the execution time for each run to a file.

We obtain the following graph:

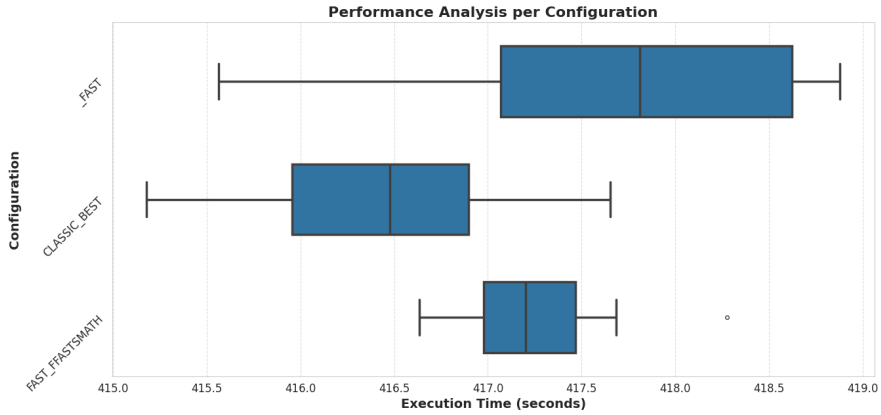

Figure 5: Configuration Times

- The `FAST` configuration has the highest median execution time, with a wide interquartile range (IQR), indicating high variability in performance. The presence of long whiskers suggests that some runs were significantly faster or slower than the median
- The `CLASSIC_BEST` configuration has the lowest median execution time and also has a large variability, but its upper whisker is shorter, meaning that extreme slowdowns are less frequent
- The `FAST_FASTMATH` configuration exhibits a lower median execution time compared to FAST and the smallest variability among all three configurations. This suggests that it is the most stable option in terms of execution time. The presence of an outlier in the `FAST_FASTMATH` configuration indicates that there was at least one unusually slow execution, but overall, it remains the fastest and most consistent option

We chose to consider the classic best as the best sequential time because it performed better than the other two configurations. This resulted in an average execution time of 416 seconds.

7 OpenMP

OpenMP allows adding directives in the code that instruct the compiler to automatically generate and manage multiple threads. These threads simultaneously execute different parts of the code, taking advantage of multiple CPU cores in a shared memory environment, aiming to optimize performance.

To parallelize the code, we added this before the outer loop:

```
[...]
    #pragma omp parallel for shared(image) schedule(dynamic)
    for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
    {
       [...]
    }
[...]
```

Listing 1: OpenMP code in C

- `#pragma omp parallel for` → This directive indicates that the outer loop will be executed in parallel
- `shared(image)` → This variable is shared among all threads, allowing them to access the same data in memory
- `schedule(dynamic)` → There are several types of schedulers for OpenMP; from our previous lab tests, we found that the dynamic scheduler was the most efficient in that case. This type of scheduler follows a *first come, first served* approach. This technique is particularly advantageous when the workload is unpredictable, as in this case. Initially, each thread receives a single data block; once completed, the thread requests a new block, thus ensuring more efficient resource utilization. After performing measurements, we confirmed that in this algorithm (as the number of iterations can vary from cell to cell), it performs better than both static and guided scheduling

7.1 Performance time evaluation

To visualize performance differences, we decided to run the code using a different number of threads. We chose to consider the following thread numbers: 1, 2, 4, 8, 12, 16, 20, 40.

- **1** → Sequential run
- **2, 4, 8, 16** → To gather more data when increasing the number of threads
- **12** → Equal to the number of physical cores
- **20** → The maximum number of threads
- **40** → To observe the overhead

The code is executed 5 times for each specified number of threads, to obtain reliable average values for optimal analysis, resulting in the following chart:

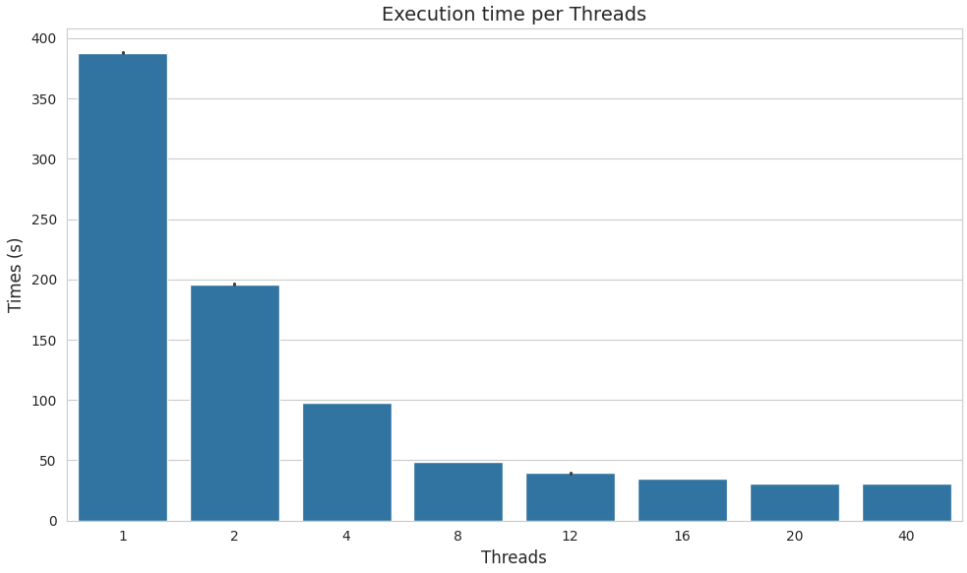

Figure 6: Bar chart OpenMP

Num Threads	Average Time (s)
1	387.91
2	195.97
4	97.88
8	49.03
12	39.45
16	34.98
20	31.16
40	31.18

Table 2: Average time table (s) for each number of threads

Analyzing the chart 6 and the table above, we can state that the best time is achieved when using the maximum number of threads that the CPU allows simultaneously. When exceeding the maximum number of available threads, in this case, 20, we observe overhead.

7.2 Speedup and efficiency

Figure 7 shows the comparison between actual speedup and ideal speedup. The speedup is defined as:

$$S = \frac{T_1}{T_P}$$

where T_1 is the average serial execution time and T_P is the average time with P threads.

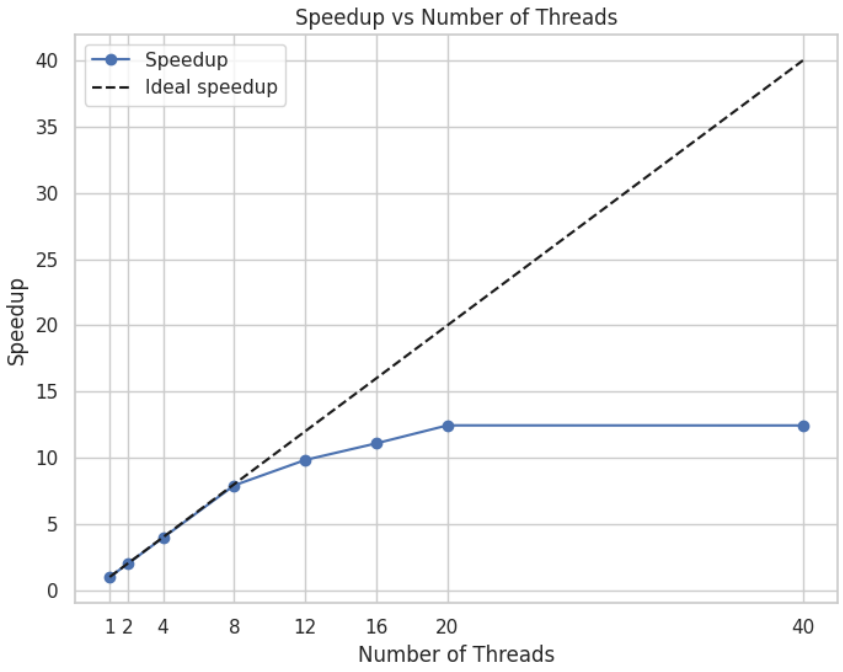

Figure 7: Speedup OpenMP

We observe that:

- The speedup increases with the number of threads, but not in a perfectly linear fashion. This indicates that performance gains increase with more threads but do not reach the ideal behavior due to factors such as thread management overhead and scalability limits
- For a number of threads less than 12 (equal to physical cores), the speedup approaches linearity. However, for $P > 12$, the speedup increase slows down, suggesting that hyper-threading introduces lower efficiency compared to increasing physical cores

- For 20 and 40 threads, the speedup stabilizes, indicating the presence of overhead and resource saturation

Figure 8 shows the comparison between actual and ideal efficiency as a function of the number of threads. Efficiency is calculated as:

$$E = \frac{S_P}{P}$$

where S_P is the speedup and P is the number of threads used.

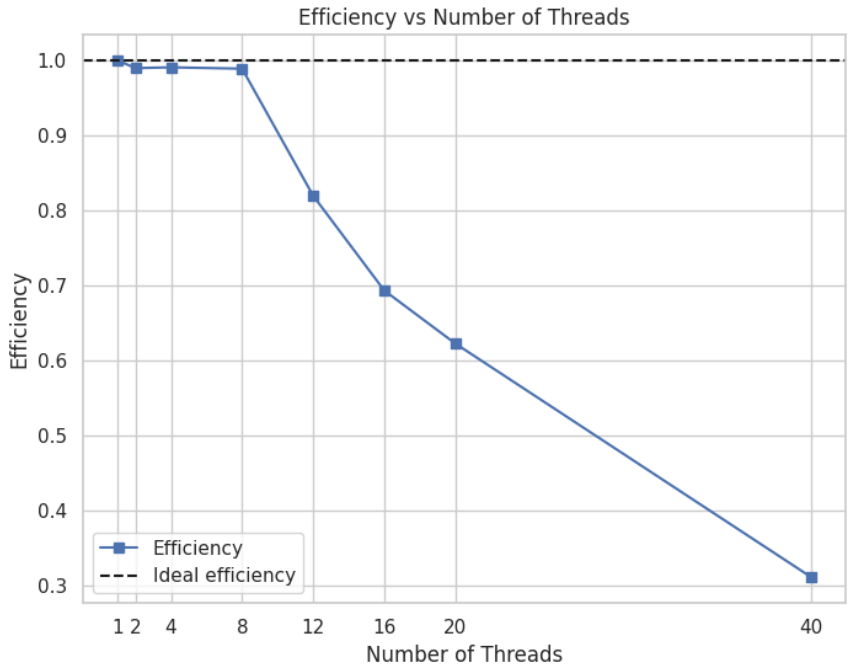

Figure 8: Efficiency OpenMP

We observe that:

- Efficiency decreases as the number of threads increases. Up to 8 threads, the system maintains relatively high efficiency, close to the ideal value
- After 8 threads efficiency start to drop significantly, indicating that thread management overhead and resource contention impact performance
- With 40 threads, efficiency is about 1/3 of the ideal value, confirming that a high number of threads does not lead to a proportional performance increase due to competition for available resources

Therefore, the maximum performance improvement is obtained up to the number of available physical cores. Beyond this value, increasing the number of threads does not bring significant advantages in terms of speedup and reduces the overall efficiency of the system.

8 CUDA

To create the `mandelbrot.cu` file, we made several changes to the original `mandelbrot.cpp` code.

The first major change is adding support for memory allocation and management on the GPU. In the original code, data was stored exclusively in system memory (RAM), while in the `mandelbrot.cu` file, data is transferred to the GPU memory to take advantage of the parallelism offered by CUDA. To do this, we used the `cudaMalloc` function to allocate memory on the GPU and the `cudaMemcpy` function to copy data from GPU memory back to RAM, as shown in the code snippet below.

```
int THREAD_SIDE = atoi(argv[1]);

int *const image = new int[c * WIDTH];

int* image_device;
cuda_check(cudaMalloc(&image_device, HEIGHT * WIDTH * sizeof
    (int)));

[...]
dim3 threads(THREAD_SIDE, THREAD_SIDE);
dim3 blocks((WIDTH + threads.x - 1) / threads.x, (HEIGHT +
    threads.y - 1) / threads.y);

[...]
// Call the kernel
mandelbrot<<<blocks,threads>>>(image_device);

cuda_check(cudaMemcpy(image, image_device, HEIGHT * WIDTH *
    sizeof(int), cudaMemcpyDeviceToHost));
cuda_check(cudaFree(image_device));
```

Listing 2: Memory Allocation and Swapping in CUDA

`THREAD_SIDE` sets the number of threads in a block. 2D blocks are leveraged, with `THREAD_SIDE` $\times$ `THREAD_SIDE` threads per block. A 2D grid contains the blocks: the number of blocks in the two grid dimensions is computed by dividing `HEIGHT` and `WIDTH` by `THREAD_SIDE`, respectively, and rounding up.

Another change was to extract the `for` loop outside the main function and create a separate kernel function `mandelbrot`. This way, it is possible to perform this operation in parallel on the GPU. The function is defined as `__global__` and we

replaced the `for` loop with a check on the thread index within the GPU grid, as shown in the code snippet below.

```
__global__ void mandelbrot(int *image) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (j > 0 && j <= HEIGHT - 1 && i > 0 && i <= WIDTH - 1)
        {
        int pos = j * WIDTH + i;

        image[pos] = 0;
        cuDoubleComplex c = make_cuDoubleComplex(i * STEP +
            MIN_X, j * STEP + MIN_Y);

        cuDoubleComplex z = make_cuDoubleComplex(0.0, 0.0);
        for (int k = 1; k <= ITERATIONS; k++)
        {
            z = cuCadd(cuCmul(z, z), c);
            if (cuCabs(z) >= 2) {
                image[pos] = k;
                break;
            }
        }
    }
}
```

Listing 3: `Mandelbrot` function edited

Thread indexing is achieved using some CUDA variables:

- `blockIdx` → Block coordinates within the grid
- `blockDim` → Block dimensions in thread units
- `threadIdx` → Thread coordinates within the block

All these variables have coordinates on three dimensions: x, y, z; which can be used to obtain row and column indices for the current thread running the function, in our case via variables `i` and `j`. This way, the kernel replaces the loop with threading, and each thread works on a part of the problem independently.

Additionally, we introduced functions offered by `cuComplex` to improve performance when calculating complex numbers on CUDA. The operations we considered are:

- `make_cuDoubleComplex` and `cuDoubleComplex` → For creating complex numbers using CUDA
- `cuCadd` → To calculate the sum of complex numbers using CUDA

- `cuCmul` → To calculate the multiplication of complex numbers using CUDA
- `cuCabs` → To calculate the absolute value using CUDA

8.1 Performance time evaluation

Keeping the computational complexity at 5000 for `RESOLUTION` and 5000 for `ITERATION`, we calculated the serial time T1 using this command line:

`!g++ -O3 -ffast-math /content/mandelbrot.cpp -o mandelbrot`.

Obtaining a time of 891.27 seconds. Then we used the following compiler with specific flags to compile the `mandelbrot.cu` file:

`!nvcc -O3 -arch=sm_75 /content/mandelbrot.cu -o mandelbrot`.

Next we ran the code 10 times for different values of `THREAD_SIDE` (the single dimension value of the square number of threads per block). The chosen values were 1, 2, 4, 8, 16, 32, and 64. The version with `THREAD_SIDE` 1 is the version compiled with GCC using the CPU, as a single GPU thread performed significantly worse.

THREAD_SIDE	**Average time (s)**
1	891.27
2	439.77
4	111.04
8	55.91
16	56.23
32	56.69
64	0.47

Table 3: Time table

From the table above, it can be observed that, with a `THREAD_SIDE` value of 64 (resulting in a total number of threads per block of 4096), the average execution time is completely out of scale. Checking the output of the Mandelbrot calculation, the result is a matrix full of zeros, suggesting that the calculations are either not performed correctly or some issue occurs. This happens because $64 \times 64 = 4096$, which exceeds the maximum number of threads per block (1024). For this reason, we decided not to consider this specific `THREAD_SIDE` value in the next plots.

After creating this table, we visualized the results with a bar chart.

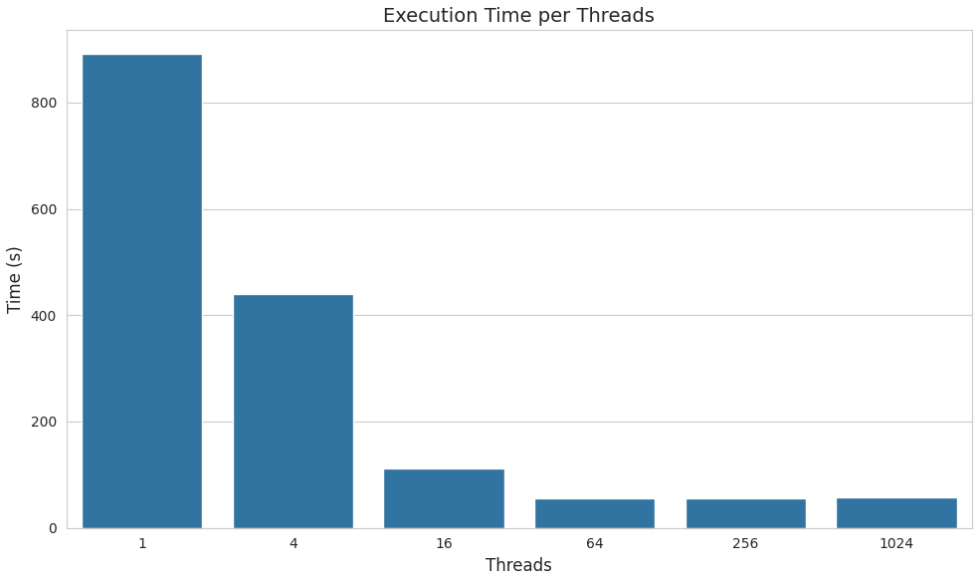

Figure 9: Time plot CUDA

Analyzing the chart 9 and the table above, the best time is obtained with a `THREAD_SIDE` value of 8 (64 threads). Increasing this value causes the bars to flatten, so it is not useful to allocate more resources. This suggests that the best balance is achieved between 16 and 64 threads per block; beyond this threshold, hardware resources such as memory and thread management begin to limit performance gains.

8.2 Profiling

To evaluate the performance of the CUDA code, we used the `nvprof` command to obtain a detailed analysis of GPU usage.

THREAD_SIDE	**Kernel Time (s)**	**Memcpy DtoH (s)**	**Total Time (s)**
2	438.87	0.43	439.3
4	110.13	0.43	110.56
8	55.47	0.53	56
16	55.64	0.42	56.07
32	56.09	0.42	56.51

Table 4: Profiling CUDA

From Table 4, we can observe that:

- The total execution time is dominated by the execution time of the mandelbrot kernel
- The optimal value of `THREAD_SIDE` appears to be between 8 and 16, since beyond this value the performance improvement becomes marginal

8.3 Speedup and Efficiency

Remembering that the formula is:

$$S = \frac{T_1}{T_P}$$

where T_1 is the average serial execution time and T_P is the average time with P threads.

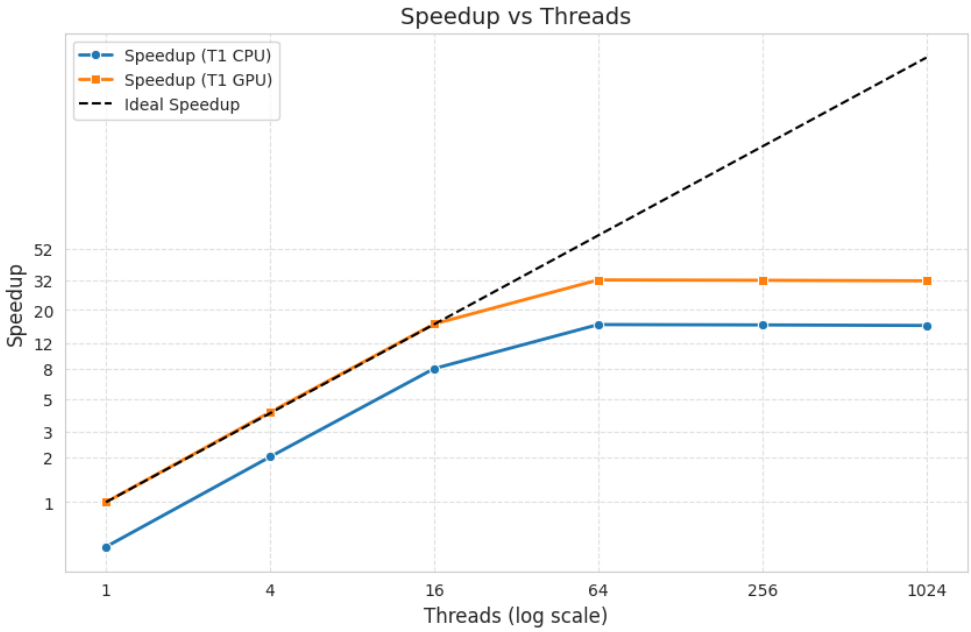

Figure 10: CUDA Speedup

The first graph shown in Figure 10 (with logarithmic scales on both axes) displays the behavior of the speedup in relation to the number of threads per block. We decided to display both the T1 CPU and T1 GPU versions simultaneously, always referring to the times calculated with Colab.

In this case, it follows a common and typical behavior, as observed in the OpenMP case; the measured speedup initially tends to be similar to the ideal speedup up to a certain range $16-64$; and then converges to a lower value. This

means that the performance increase is not directly proportional to the number of threads employed.

Remembering that Efficiency is calculated as:

$$E = \frac{S_P}{P}$$

where S_P is the speedup and P is the number of threads used.

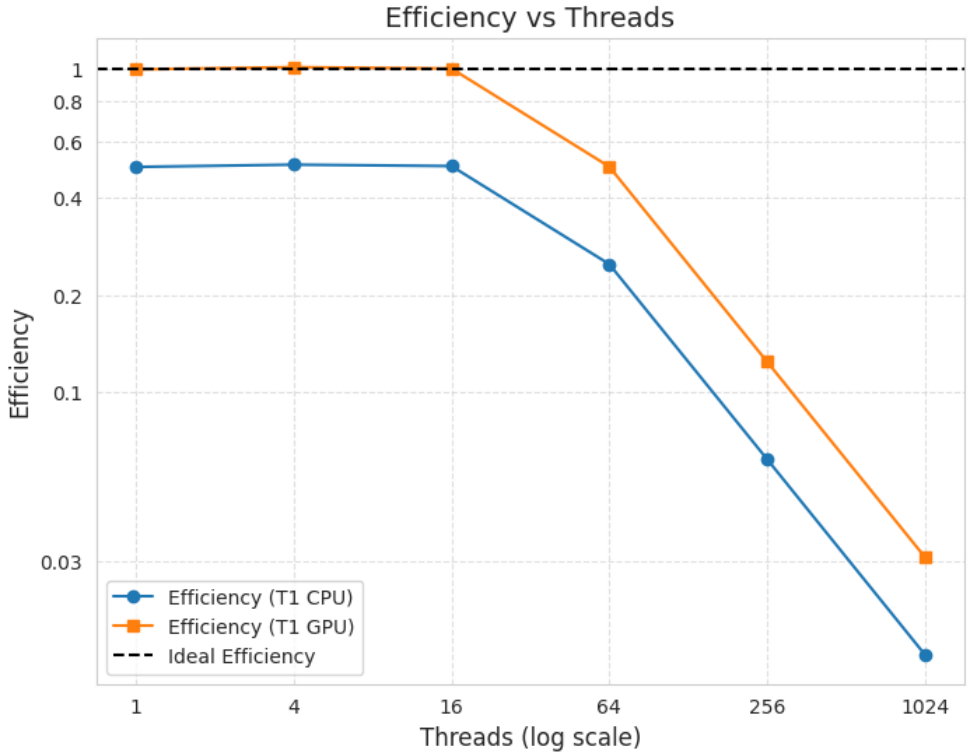

Figure 11: CUDA Efficiency

Even in this case, we decided to display both the T1 CPU and T1 GPU versions simultaneously, always referring to the times calculated with Colab. In the second graph 11, (again with logarithmic scales on both axes) the behavior of the efficiency as a function of the number of threads per block is shown. Again, as in the case of OpenMP efficiency, we observe a common and typical case where the efficiency initially follows the optimal behavior and then decreases drastically as the number of threads increases.

9 Compare OpenMP and CUDA

Once the detailed analysis of the two cases was completed, we decided to compare them more thoroughly in this section 9. Recalling that the resolution and the iteration are 5000, and the best sequential time on average is 416 seconds, we have created the summary table below.

Method	**Best thread**	**Avg best time(s)**	**Speedup**	**Efficiency**
OpenMP	20	31.16	12.44	0.622
CUDA	8×8	55.91	15.93	0.24

Table 5: Compare table

The table contains the best times and the number of threads for both methods considered, and only for these two best cases, the speedup and efficiency. From the table it can be noted that we have a lower average time using OpenMP, in this specific case, with the number of threads equal to the maximum available on the system. The CUDA speedup is higher but the OpenMP efficiency turned out to be higher than CUDA's, suggesting that the CPU is able to better exploit the available resources, probably due to reduced communication latency. On the other hand, CUDA has demonstrated that the number of threads per block must be optimized to avoid overhead and ensure maximum GPU utilization. For values greater than 16 threads per block, there is a reduction in performance gains.

10 Conclusion

This report has analyzed various methods to optimize the calculation of the Mandelbrot set using OpenMP and CUDA, evaluating the performance of each approach through tests, comparison metrics, and graphical visualizations.

Theory suggests that OpenMP is more suitable when working with a limited number of cores but with fast shared memory. It proves to be more efficient in operations that are less dependent on massive parallelism and when communication between computational units must be minimized. In contrast, CUDA is advantageous when the problem is highly parallel and the GPU can execute thousands of operations simultaneously. However, memory setup and code management are more complex compared to OpenMP. A disadvantage of CUDA is the need to transfer data between the CPU and GPU, an operation that introduces overhead and limits the benefits of hardware acceleration.

In the initial phase of the report, we identified the main hotspots, the operations that consumed the most time, and we experimented with various compiler configurations to reduce these costs. The graphs related to execution times and configuration analyses provided valuable insights into the impact of compiler flags.

From our analyses, we verified that OpenMP achieved better performance for our specific case, with a lower execution time and higher efficiency compared to CUDA. Furthermore, OpenMP proved to be more scalable in our scenario, with optimal utilization of physical cores and efficient thread management.

From the graph visualizations, we observed that for OpenMP, increasing the number of threads led to a significant reduction in execution times up to the number of available physical cores. Beyond that threshold, performance plateaus, as reflected by a decrease in efficiency, as evidenced by the speedup and efficiency graphs.

In contrast, in CUDA the graphs showed that increasing the `THREAD_SIDE` parameter beyond the supported limit (1024 threads per block) leads to unreliable results, while an optimal choice (around 16-64 threads per block) guarantees competitive execution times, although still higher than the OpenMP approach.

These observations highlight that the choice of parallelization technology must be closely linked to the characteristics of the problem and the available hardware architecture. In our scenario, the ease of implementation and efficient memory management made OpenMP the most suitable solution for computing the Mandelbrot set. In conclusion, OpenMP proved to be the best choice for our Mandelbrot implementation, while CUDA could offer advantages in applications with very high and highly parallel computational loads.

List of Figures