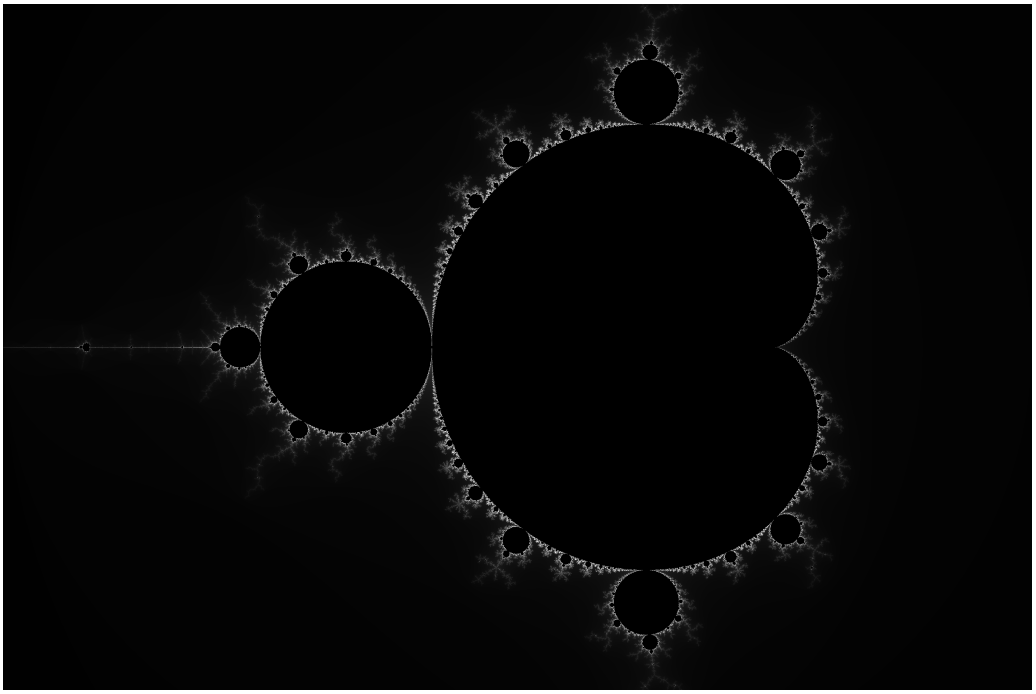
High Performance Computing
Final project
Mandelbrot optimisation

Federico Fontana
s4835118@studenti.unige.it

Filippo Siri
s4819642@studenti.unige.it

Marco Zucca
s4828628@studenti.unige.it

May 2024



Contents

1 Introduction

This project aims to optimize the implementation of an algorithm that generates the Mandelbrot set. To do so, we first analyze the performance characteristics of the algorithm by just adding OpenMP to the baseline implementation and then modifying the program to use intrinsics that allow us to increase its performance by 300%. Finally, we analyze the characteristics of the same reference algorithm running on an NVIDIA GPU, compare it to our fastest CPU implementation, and understand which is faster and why.

2 Setup

Our algorithm implementations are optimized specifically for a machine equipped with:

- one 12^{th} Gen Intel(R) Core(TM) i9-12900K CPU. It has 24 logical cores, split into 8 physical 'e-core's (no hyperthreading) and 8 physical 'p-core's (hyperthreading factor 2). All the cores support the Intel AVX2 extension, which boasts 256-bit registers, and three operand fused-multiply-add (FMA3).
- one NVIDIA T400 GPU

2.1 Metrics and evaluation

Each execution time result we show is an average of 10 runs with the same parameters unless stated otherwise. To evaluate our optimizations, we computed two different metrics starting from the average time:

- **Speedup**: is the value expressing the performance improvement over the single thread implementation.

$$Speedup_{avg}(NumThreads) = \frac{AverageTime_{\#Thread=1}}{AverageTime_{\#Thread=NumThreads}}$$

- **Efficiency**: is the value expressing how much speedup we gain by adding threads.

$$Efficiency_{avg}(NumThreads) = \frac{Speedup_{avg}(NumThreads)}{NumThreads}$$

3 Hotspots

Before optimizing the algorithm, we looked at the code to identify where most of the computing is done. Since the code is quite simple, it is easy to identify the following hotspots:

```
for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
{
    image[pos] = 0;

    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);

    // z = z^2 + c
    complex<double> z(0, 0);
    for (int i = 1; i <= ITERATIONS; i++)
    {
        z = pow(z, 2) + c;

        // If it is convergent
        if (abs(z) >= 2)
        {
            image[pos] = i;
            break;
        }
    }
}
```

Listing 1: Reference algorithm in C

Profiling the binary with Intel Advisor, we get the roofline model and the call stack information shown in Figure 1. More precisely, from the roofline we can confirm that the hotspot is indeed the one shown in listing 1, and since the roofline model shows a green circle, the code that comprises it cannot be optimized more than that in its current form.

As confirmed by the roofline in Figure 1, the hotspot in listing 1 is the only hotspot in the program, and as such we will only be focusing on optimizing it.

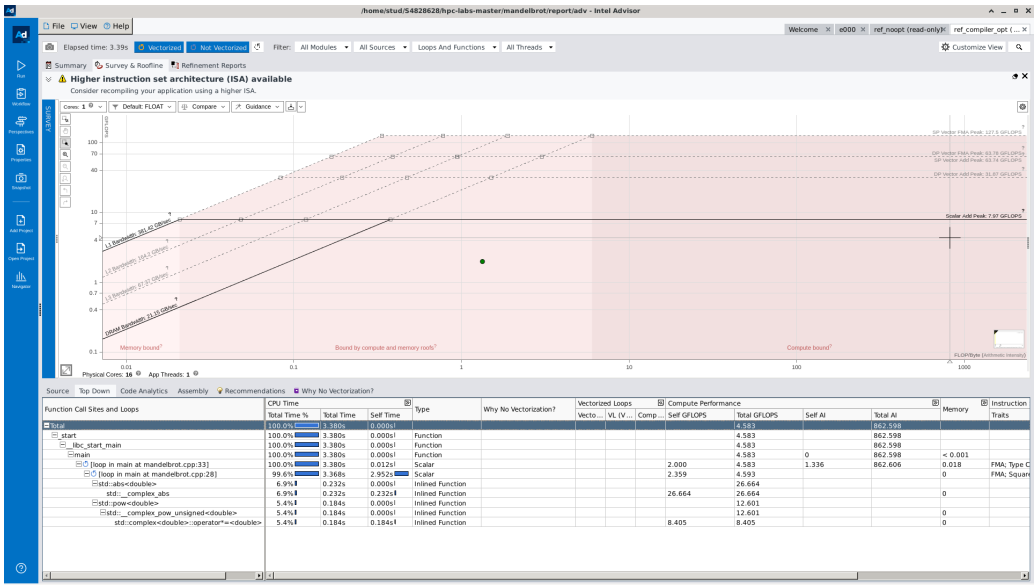


Figure 1: Screenshot of Intel Advisor profiling the reference algorithm

Compiling the reference implementation with `icpx` and the `-fast -xHost -qopt-report=max` flags, we notice that the compiler is not able to automatically vectorize the code, as shown in listing 2.

```
...
LOOP BEGIN at src/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 src/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 (1333:13) and (1350:13)
        remark #15346: vector dependence: assumed FLOW dependence
            between (1333:13) and (1350:13)
        remark #15346: vector dependence: assumed FLOW dependence
            between (1333:13) and (1350:13)
        remark #15346: vector dependence: assumed FLOW dependence
            between (1333:13) and (1350:13)
        remark #15346: vector dependence: assumed FLOW dependence
            between (1333:13) and (1350:13)
        remark #15346: vector dependence: assumed FLOW dependence
            between (1333:13) and (1350:13)
        remark #25438: Loop unrolled without remainder by 2
    LOOP END
LOOP END
...
```

Listing 2: Hotspot section of the vectorization report produced by `icpx` when compiling the baseline implementation

Two main reasons stop the compiler from automatically vectorizing the program:

- the calculations performed in the inner loop depend on the results of the previous iteration, as shown in listing 2
- the inner loop contains a `break`: this makes the number of iterations non-constant and varies according to a condition verified at runtime, which stops the compiler from vectorizing the code
- for each value `pos` the code creates some variables to be modified by the inner loop, meaning that the loops cannot be trivially swapped by the compiler

The algorithm can however be vectorized by slightly changing the logic and using the intrinsics available in the `immintr.h` C header, as we will see in section 4.3.

4 CPU

4.1 Initial optimizations

The first optimization we decided to perform is a simple change in the guard controlling the `break` in the inner loop: `abs(z) >= 2` can be safely written as `norm(z) >= 4`, removing the heavy computation of the square root. This simple yet effective optimization brings a performance increase of around 1.4-2% with respect to the reference implementation, as reported in table 1, and as such it will be included in all future versions of the program we will analyze in the next sections.

Resolution	Reference	Optimized	Time saved (%)
500	849	837	1.38
1000	3391	3347	1.31
1500	7627	7464	2.13
2000	13559	13357	1.49

Table 1: Average execution times with and without the `norm` optimization. Execution times are shown in `ms`.

4.2 Parallelisation

As mentioned in section 3, the outer loop of the reference implementation is the only one that doesn't show dependencies between one loop and the results of the previous one. For this reason, we decided to parallelize it by using OpenMP's directives. Furthermore, we decided to parametrize the `schedule` and `num_threads` parameters to analyze them in the following sections, to find the most performant configuration on our CPU.

The modified source code of the hotspot is shown in listing 3.

```
#pragma omp parallel for schedule(OMP_SCHEDULE) num_threads(THREAD_NO)
for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
{
    image[pos] = 0;

    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);

    // z = z^2 + c
    complex<double> z(0, 0);
    for (int i = 1; i <= ITERATIONS; i++)
    {
        z = pow(z, 2) + c;

        // If it is convergent
        if (norm(z) >= 4)
        {
            image[pos] = i;
            break;
        }
    }
}
```

Listing 3: Parallelised code with OpenMP

4.2.1 OpenMP schedule choice

The main parameter we can tweak in this implementation is the OpenMP schedule, which governs how the tasks are split and distributed to the worker threads. We suspect that the `dynamic` OpenMP schedule will lead to better performance compared to the default `static` schedule for two main reasons:

- **CPU architecture:** our CPU has different types of cores with different performance characteristics, meaning that the same task could be completed faster or slower depending on the physical core on which it is executed
- **Uneven tasks:** Some tasks in which the result will be split require more time to be computed. For example, with the default values for `MIN_X`, `MAX_X`, `MIN_Y`, `MAX_Y`, `RESOLUTION`, and `ITERATIONS` present in the reference implementation, the first row of the result is composed of numbers close to 0, meaning that

the inner loop requires only a few cycles to complete, while numbers close to the core of the Mandelbrot set diverge after many more iterations, meaning that the chunks containing them take much more time to compute, even on the same CPU core type.

Resolution	500		1000		1500		2000	
Scheduler / #Threads	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static
1	799	809	3166	3205	7092	7115	12645	12764
2	406	405	1617	1605	3623	3598	6455	6385
4	204	334	811	1323	1820	2953	3243	5278
8	104	204	411	800	916	1780	1630	3206
16	67	128	257	507	562	1111	999	1897
24	56	100	208	376	461	812	812	1458

Table 2: Execution time comparison between `static` and `dynamic` schedule. Execution times are shown in `ms`.

We also tried changing the chunk size when using the `dynamic` schedule, but in the configurations we tested we found it to have no significant effect on average execution time.

All of the results shown in the report from now on will use the `dynamic` schedule when applicable.

4.2.2 Results

The results shown in this section are aimed at studying the relationship between higher resolution, the increasing number of threads used, and the metrics outlined in 2.1. Furthermore, changing the resolution and fixing the bounds of the generated image allows us to understand how the optimizations we make to the code alter the algorithm's arithmetic stability and precision.

Figure 2 shows that the average execution time increases with a larger resolution. Using more threads shows a significant performance improvement until 8 threads when the speedup increases at a slower rate.

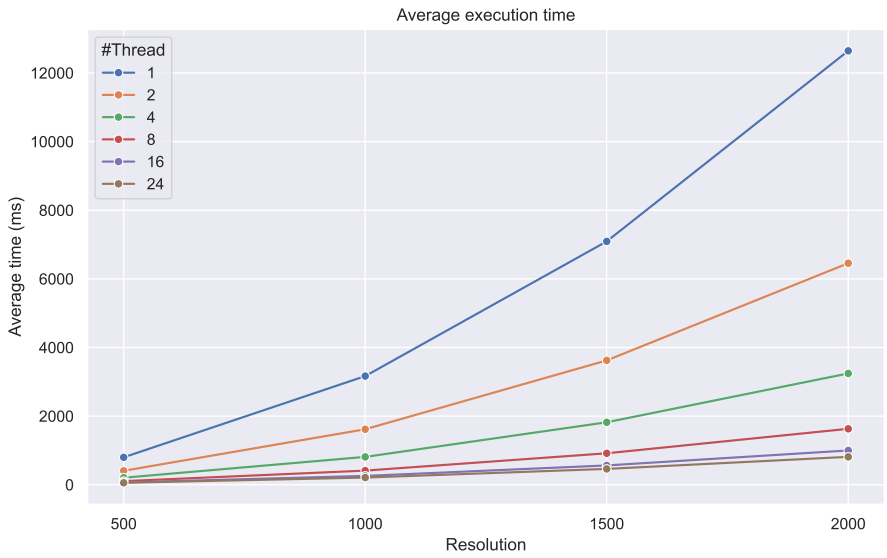


Figure 2: Average time with different number of threads and resolution

This behavior can be better observed in the speedup (Figure 3) plot, where we can see a speedup close to linear until we reach 8 threads, where it starts to show a sub-linear trend.

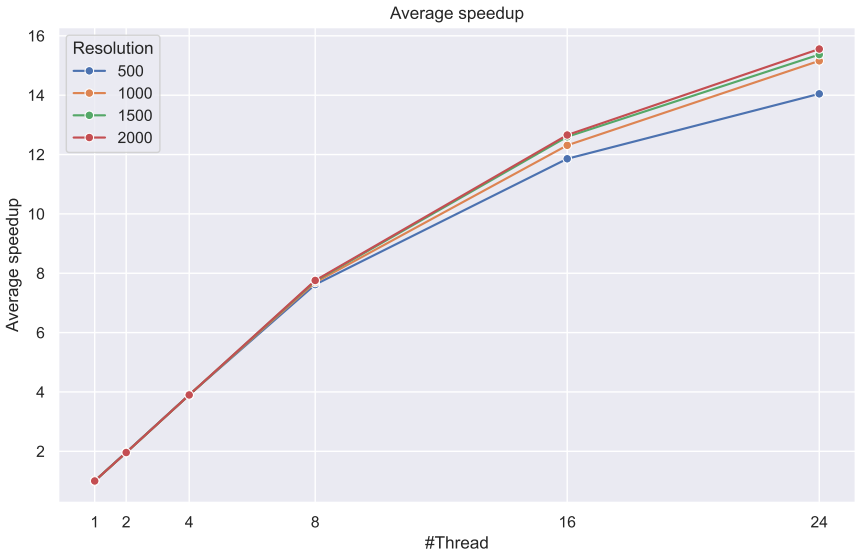


Figure 3: Average speedup with different number of threads and resolution

Furthermore, the same behavior is reflected in the efficiency plot (Figure 4), where efficiency is close to 1 until we reach 8 threads. After that, the same plot shows a significant loss in efficiency from the 8-thread mark onwards, reaching as low as around 0.65 with 24 threads.

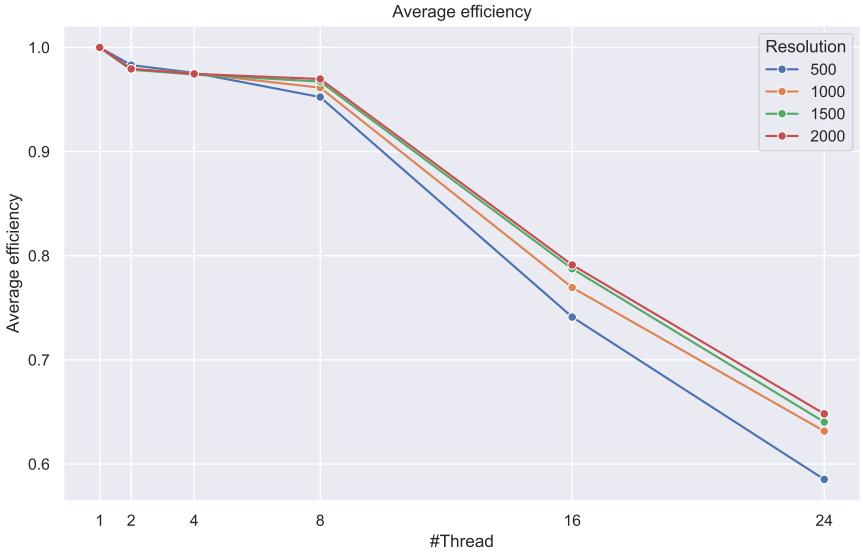


Figure 4: Average efficiency with different number of threads and resolution

Resolution	#Thread	Time (ms)	Speedup	Efficiency
500	1	799	1.00	1.00
500	2	406	1.97	0.98
500	4	204	3.90	0.98
500	8	104	7.62	0.95
500	16	67	11.86	0.74
500	24	56	14.05	0.59
1000	1	3166	1.00	1.00
1000	2	1617	1.96	0.98
1000	4	811	3.90	0.98
1000	8	411	7.69	0.96
1000	16	257	12.31	0.77
1000	24	208	15.16	0.63
1500	1	7092	1.00	1.00
1500	2	3623	1.96	0.98
1500	4	1820	3.90	0.97
1500	8	916	7.74	0.97
1500	16	562	12.60	0.79
1500	24	461	15.37	0.64
2000	1	12645	1.00	1.00
2000	2	6455	1.96	0.98
2000	4	3243	3.90	0.97
2000	8	1630	7.76	0.97
2000	16	999	12.66	0.79
2000	24	812	15.56	0.65

Table 3: Execution time, speedup, and efficiency comparison by varying the resolution and the number of threads.

4.3 Vectorization

The Intel compiler cannot automatically vectorize the operations in the hotspot of the program because of the reasons seen in section 3.

One approach aimed to aid the compiler in vectorizing the hotspot would be to add two parallel arrays to `image`, one containing the current `z` values, and one storing the values of `c`, removing the need for local variable instantiation in the outer loop. This change in operations would also require changing the logic of the if guarding the `break` statement. By implementing the aforementioned changes, the compiler could be able to swap the two loops and vectorize the code, but its performance improvements could clash with the increased use of memory to store the parallel arrays, which would also require more memory accesses which could thus decrease performance if the code is not correctly optimized.

For these reasons we decided to solve the problem with a different approach: the hotspot can be vectorized by using the C intrinsics to express the more complex relationships we know about the algorithm that the compiler can't automatically pick up. In particular, this will let us manually manage the if condition guarding the `break`.

4.3.1 Algorithm

In the hand-vectorized implementation of the program using intrinsics, the outer loop iterates over blocks of points which will be stored in the 256-bit registers provided by the CPU. The chunk sizes vary depending on the floating point type used, with effects in both performance and precision, analyzed in section 4.3.2.

To correctly vectorize the algorithm, the code can't use `c++`'s `complex` type. The real and imaginary parts of the variables `z` and `c` relative to the points in the current block are stored in four 256-bit registers, and the operations on them have been rewritten using intrinsics.

One important optimization we made to minimize memory accesses is to save only the final values (stored in the `results` variable) to RAM in the `image` array only when all the points diverged or the maximum number of iterations has been reached. Since all values in the result vector start from 0 and don't rely on any other

points' partial or final values to be computed, this means that our optimized algorithm doesn't load any value from RAM at the beginning of the body of the outer loop and only saves the values of the whole vectorized block once, when all of the points in the block have diverged, at the end of the outer loop body, outside of the inner loop's body. The effect of the loads and repeated stores in RAM could have been alleviated by the cache, but this optimization skips most of them entirely.

The condition guarding the `break` has been completely rewritten to adapt it to the vectorized code, meaning that the `break` is reached only when all the values in the current block have diverged. This means that some points will stay in the inner loop without breaking nor updating the value in the result vector and could thus reduce the peak performance of our algorithm, but we found that usually most blocks are comprised of points whose iteration counts before diverging are quite similar, with the exceptions of the blocks covering the edges of the set, which are in general few concerning the total count. Furthermore, since this is a vectorized implementation we have to keep in mind that 4 or 8 points are being processed at a time in each block depending on the floating point type used, and thus the performance should be much better than the non-vectorized implementations, as we will see in section 4.3.4.

Since some points diverge faster than others we needed to add logic to stop updating the values of the `results` variable once the point diverged. To this end, we introduced the variable `mask` where `0...0` is stored if the point has already diverged in a previous iteration and thus shouldn't be updated anymore, and `1...1` in case it hasn't diverged yet.

The first operation performed after computing the result of $z = z^2 + c$ is to update the value of the `results` variable following the pseudo code shown below. The pseudo-code assumes that the code in it is run for each value of `pos` in the range $[0, 4)$ or $[0, 8)$ in case the algorithm is using `float`s or `double`s respectively.

```
results[pos] = results[pos] | (abs2_gt_4[pos] & mask[pos] & current_step[pos])
```

The first operand of the *bitwise or* will ensure that the previous value is kept in case it doesn't need to be updated, while the second saves the current value of `i` (stored in `current_step`) if the current point diverged (saved as a mask in the variable `abs2_gt_4`) and it hasn't been updated yet (i.e. `mask[pos] == 1....1`).

Immediately after computing the new value of `results`, the mask is updated following the logic shown in the pseudo-code below. The same notes about `pos` explained above apply.

```
mask[pos] = mask[pos] & (mask[pos] ^ abs2_gt_4[pos])
```

The first operand of the *bitwise and* ensures that the value of `mask[pos]` can't switch back from all zeros to all ones after the first time it is updated, while the *bitwise xor* in the second operand makes the mask switch from all ones to all zeros when the point diverges.

The last portion of code in the body of the inner loop uses a call to the intrinsic `_mm256_movemask_pd`[1], which fills the least significant bits of the result (`dst`) with the value of the least significant bit of each value in the register passed in as input (`a`), as described in the following pseudo-code:

```
FOR j := 0 to 3
    i := j*64
    IF a[i+63]
    dst[j] := 1
    ELSE
    dst[j] := 0
    FI
ENDFOR
dst[MAX:4] := 0
```

The if check guarding the `break` statement compares the mask produced by `_mm256_movemask_pd` called on `abs2_gt_4` with `0xFF` or `0xF`, depending on the floating point type used. If they are equal all points in the block have diverged and the inner loop can stop, as continuing until `i` reaches `ITERATIONS` would yield no changes to `results`.

[1] Intel intrinsics documentation: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=5674,4605&text=_mm256_movemask_pd`

```
__m256 step = _mm256_set1_ps(STEP);
__m256 min_x = _mm256_set1_ps(MIN_X);
__m256 min_y = _mm256_set1_ps(MIN_Y);

for (int pos = 0; pos < HEIGHT * WIDTH; pos += 8)
{
    __m256 c_re = _mm256_set_ps(
        (pos + 7) % WIDTH, (pos + 6) % WIDTH, (pos + 5) % WIDTH,
        (pos + 4) % WIDTH, (pos + 3) % WIDTH, (pos + 2) % WIDTH,
        (pos + 1) % WIDTH, (pos + 0) % WIDTH);
    c_re = _mm256_fmadd_ps(c_re, step, min_x);

    __m256 c_im = _mm256_set_ps(
        (pos + 7) / WIDTH, (pos + 6) / WIDTH, (pos + 5) / WIDTH,
        (pos + 4) / WIDTH, (pos + 3) / WIDTH, (pos + 2) / WIDTH,
        (pos + 1) / WIDTH, (pos + 0) / WIDTH);
    c_im = _mm256_fmadd_pd(c_im, step, min_y);

    __m256 z_re = _mm256_setzero_ps();
    __m256 z_im = _mm256_setzero_ps();

    __m256i results = _mm256_setzero_si256();
    __m256i mask = _mm256_set1_epi32(-1);

    for (int i = 1; i <= ITERATIONS; i++)
    {
        // z2 = z * z; z = z2 + c
        __m256 z2_re = _mm256_fmsub_ps(z_re, z_re, _mm256_mul_ps(z_im, z_im));
        __m256 z2_im = _mm256_mul_ps(z_re, z_im);
        z_im = _mm256_fmadd_ps(_mm256_set1_ps(2.0), z2_im, c_im);
        z_re = _mm256_add_ps(z2_re, c_re);

        // abs2 = |z|2 = x2 + y2.
        __m256 abs2 = _mm256_fmadd_ps(z_re, z_re, _mm256_mul_ps(z_im, z_im));
        __m256 abs2_gt_4 = _mm256_cmp_ps(abs2, _mm256_set1_ps(4.0), _CMP_GT_OQ);

        __m256i current_step = _mm256_set1_epi32(i);

        // results[pos] = results[pos] | (abs2_gt_4[pos] & mask[pos] & current_step[pos])
        results = _mm256_or_si256(
            results,
            _mm256_and_si256(
                _mm256_and_si256( (__m256d) abs2_gt_4,  mask ),
                current_step
            )
        );

        // mask[pos] = mask[pos] & (mask[pos] ^ abs2_gt_4[pos])
        mask = _mm256_and_si256(mask, _mm256_xor_si256(mask, abs2_gt_4));

        // If all of the points in the register have diverged, then break out of the loop
        if (_mm256_movemask_ps(abs2_gt_4) == 0xFF)
            break;
    }
    _mm256_store_si256((__m256i *)&image[pos], results);
}
```

Listing 4: Vectorized algorithm with `floats` and `FMA`.

4.3.2 Floats vs doubles

Since `floats` only need 32 bits of storage as opposed to `double`'s 64-bit, the 256-bit registers available on our CPU can fit twice as many `floats`. This should cut execution time by about 50%, but in turn, it could decrease the algorithm's numeric stability.

We decided to try the various program configurations with the two types of variables to evaluate the algorithm's performance both in terms of precision and performance.

Figure 5 shows the results we were expecting in terms of performance: we found an increase in performance of about 50% when using `floats` concerning `doubles`. The raw performance data is available in table 4.

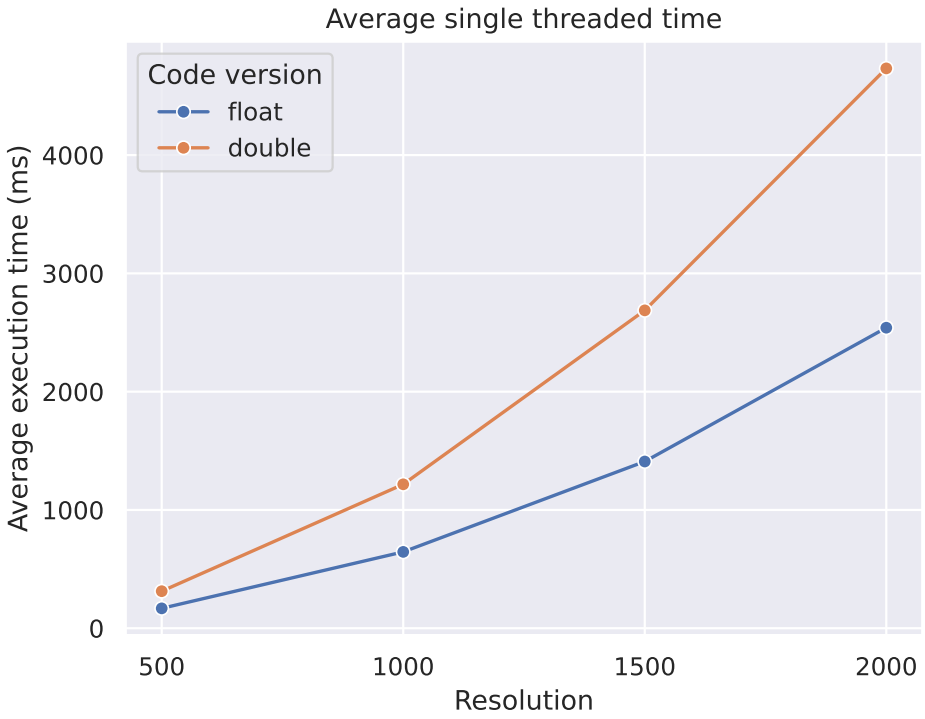


Figure 5: Average execution time with different resolutions

Resolution	Float	Double
500	168	313
1000	645	1216
1500	1409	2687
2000	2540	4732

Table 4: Execution time comparison between `floats` and `doubles`. Execution times are shown in ms.

In terms of precision, we noticed a big change in the numbers present in the final file created by the program. In particular, we noticed that for each `RESOLUTION` value we tried, the percentage of bits that differed from the version using the `doubles` to the one using `floats` is about 3%. Furthermore, by analyzing the difference between the two results in the form of an image (Figure 6) we noticed that the differences usually lie on the edge of the set where it shows its details, and the added precision of `doubles` helps in capturing them.

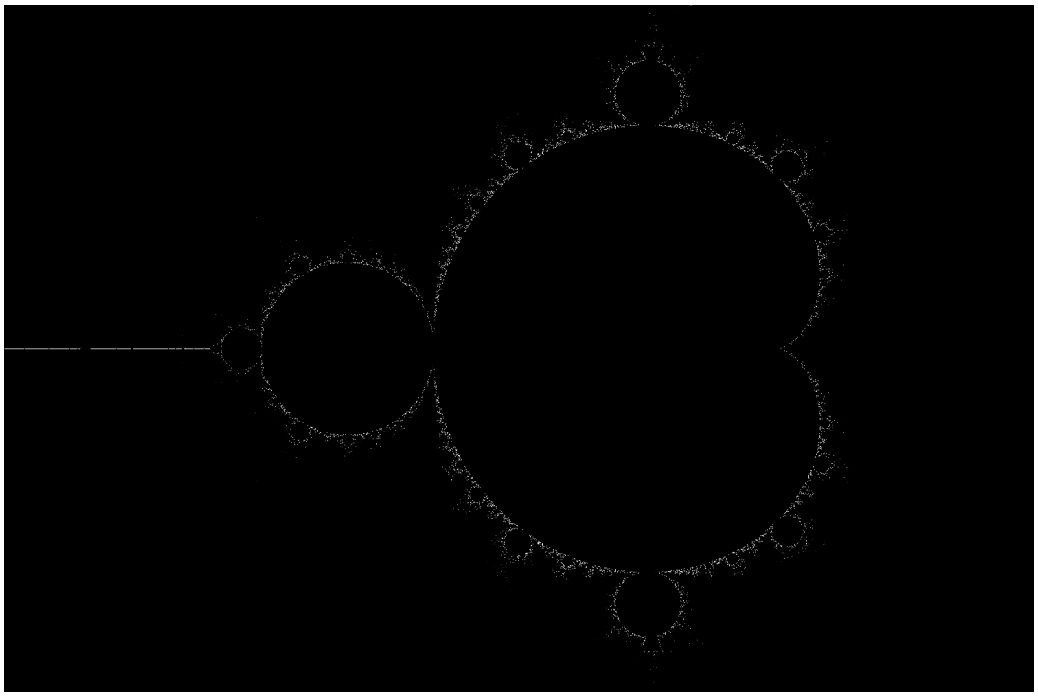


Figure 6: Difference between the images produced by using `floats` and `doubles`. Produced with the outputs of programs running with `RESOLUTION=500`

Table 5 shows more precisely the difference between the version using `floats` and `doubles`.

Resolution	Points	Difference (%)
500	7559	3.02
1000	29687	2.97
1500	53872	2.39
2000	118307	2.96

Table 5: Number of different numbers in the final result between the program using `floats` and `doubles`

Since the beauty of the set lies in its details, we decided to keep using `doubles` in the next experiments.

4.3.3 FMA

All the cores in our CPU can use FMA-style instructions, and since the code that performs the calculations in the hotspot of the program can use them, we decided to measure their impact on the performance and the correctness of the program.

We were able to use them 5 times in the code. The changes were easy to make (e.g. listing 5), but led to a small but noticeable performance increase nonetheless.

```
#ifdef FMA
    c_im = _mm256_fmadd_ps(c_im, step, min_y);
#else
    c_im = _mm256_mul_ps(c_im, step);
    c_im = _mm256_add_ps(c_im, min_y);
#endif
```

Listing 5: Example of code written using FMA operations

In particular, Figure 7 shows that FMA instructions increased the performance of the algorithm by about 15% of the base implementation over the vectorized implementation that doesn't use them.

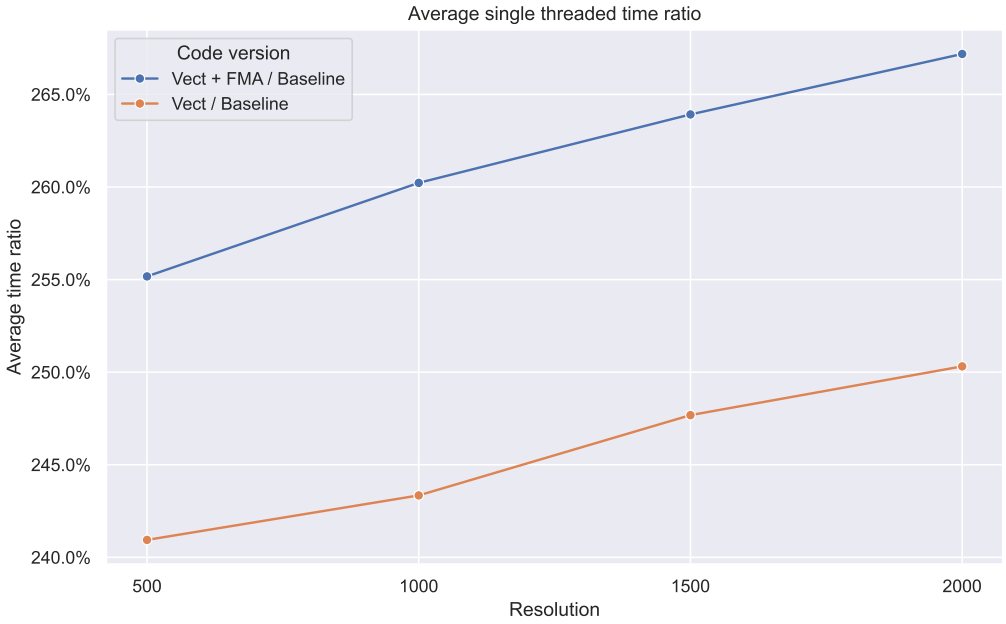


Figure 7: Single thread time and vectorized single thread ratio difference compared to `FMA` version

Resolution	Vect(%)	Vect + FMA(%)
500	240.94	255.17
1000	243.34	260.22
1500	247.68	263.92
2000	250.31	267.19

Table 6: Performance increase of single-threaded vectorized implementation relative to the reference single-threaded implementation

Furthermore, other than boosting the speed of the algorithm, FMA operations perform rounding only once before saving the final result[2], thus reducing the loss in numerical precision in the operations as opposed to performing the two separate operations. This increases the numerical stability of the algorithm and should thus lead to better qualitative results.

Since this simple optimization has no drawbacks, all the results in further comparisons involving the hand-vectorized code will be made with this optimization active (i.e. in our case the binary will be compiled with the `-DFMA` flag), unless stated otherwise.

[2]Intel Instruction Set reference, page 1671-1673

4.3.4 Results

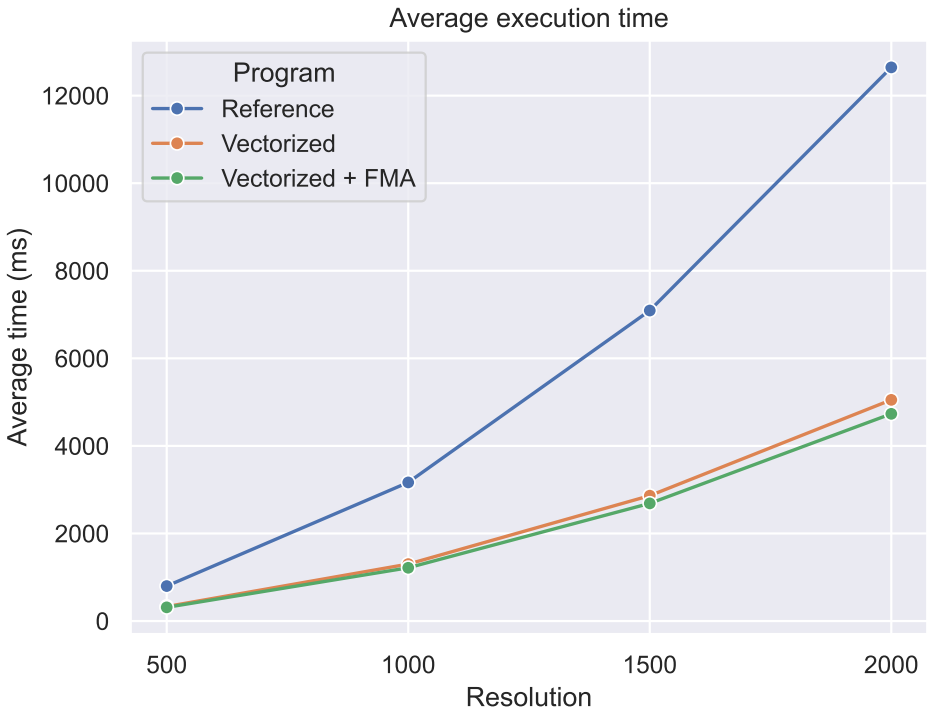


Figure 8: Execution times of various single-threaded algorithm implementations

Resolution	Reference	Vect	Vect + FMA
500	799	331	313
1000	3166	1301	1216
1500	7092	2863	2687
2000	12645	5051	4732

Table 7: Execution time comparison between various single-threaded implementations. Execution times are shown in ms

Figure 8 shows an increase in performance between the reference and the vectorized implementation. The plot also shows a slight performance increase brought by FMA, discussed in section 4.3.3.

In general, we can see that vectorizing the algorithm brings a performance increase of about 260% in the single-threaded configurations we tried, according to Figure 7. Furthermore, the added flexibility of manually deciding which CPU instructions will be issued by the binary lets us squeeze some extra performance and precision by using specialized instructions like the ones in FMA.

4.4 Vectorization + Parallelisation

In this section, we analyze the performance gains we get by adding multithreading to our hand-vectorized algorithm compared to the baseline OpenMP implementation.

4.4.1 Algorithm

The process of adding multi-threading to the vectorized algorithm only entails adding one `pragma` preprocessor instruction above the outer loop. This lets us control the schedule and the number of threads used to run the program. A summary of the code for this program version can be found in listing 6.

```
#pragma omp parallel for schedule(OMP_SCHEDULE) num_threads(THREAD_NO)
for (int pos = 0; pos < HEIGHT * WIDTH; pos += 8)
{
    ...
    for (int i = 1; i <= ITERATIONS; i++)
    {
        ...
    }
    ...
}
```

Listing 6: Summary of the vectorized and parallelized algorithm

4.4.2 Vectorization impact on performance

Before looking at the whole picture and the overall performance of the final iteration of our fastest CPU algorithm it's important to compare its performance with the parallelized reference algorithm.

Figure 9 shows that the execution times have drastically decreased. In particular, the vectorized version using 2 threads performs approximately on par with the non-vectorized version using 4 threads.

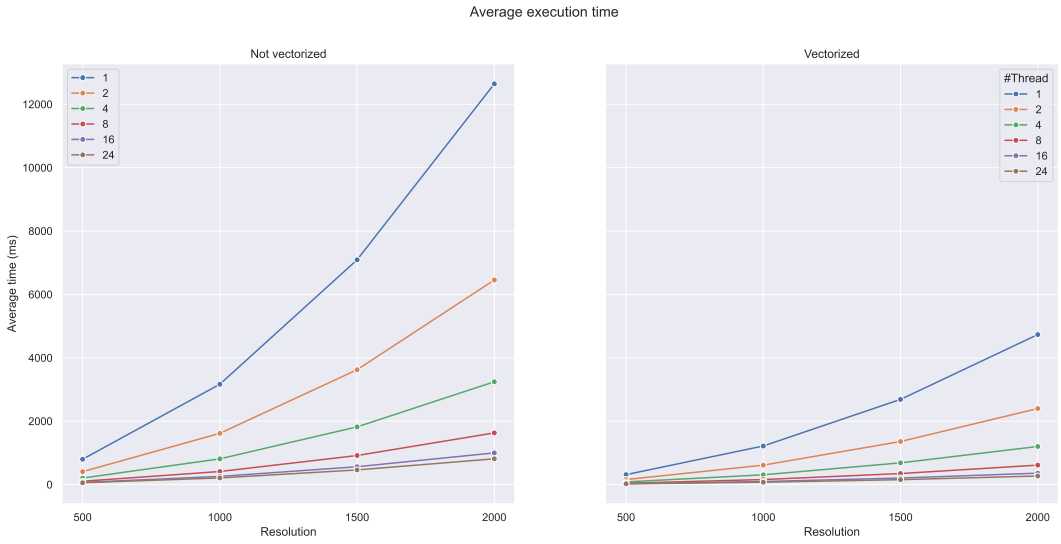


Figure 9: Comparison between average execution time trends of non-vectorized and vectorized algorithm versions

Resolution	500		1000		1500		2000	
Vectorization / #Threads	Not Vect	Vect	Not Vect	Vect	Not Vect	Vect	Not Vect	Vect
1	799	313	3166	1216	7092	2687	12645	4732
2	406	158	1617	612	3623	1358	6455	2399
4	204	83	811	307	1820	684	3243	1201
8	104	42	411	157	916	347	1630	613
16	67	28	257	95	562	205	999	359
24	56	24	208	73	461	154	812	270

Table 8: Average execution times of non-vectorized and vectorized program configurations. Execution times are shown in ms

Furthermore, analyzing the ratios between the vectorized and non-vectorized versions shown in Figure 10 highlights an interesting trend: when the program is given enough time to run with higher resolutions, the effect of

vectorization is more pronounced with a higher number of threads. In particular, the single-threaded version shows an execution time ratio of around 270%, and as the number of threads increases it approaches 300% (raw data shown in table 9), which it reaches with 24 threads and the highest resolution we tried.

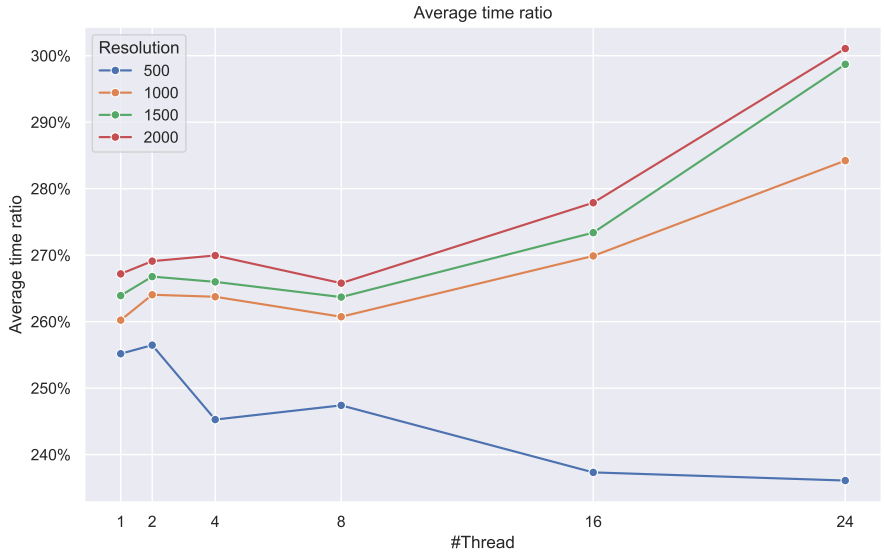


Figure 10: Time ratio (%) between the average execution time of the vectorized version and the non-vectorize version of our algorithm, with varying number of threads and `RESOLUTION`

Resolution \ #Threads	1	2	4	8	16	24
500	255.17	256.47	245.27	247.41	237.32	236.10
1000	260.22	264.04	263.74	260.73	269.88	284.22
1500	263.92	266.77	265.98	263.69	273.39	298.71
2000	267.19	269.10	269.95	265.79	277.89	301.07

Table 9: Time ratio (%) between the average execution time of the vectorized version and the non-vectorize version of our algorithm, with varying number of threads and `RESOLUTION`

We suspect that the performance characteristics of the program with `RESOLUTION=500` highlighted by Figure 10 and other plots are because the overhead of starting more than two threads has more impact than the performance gains they bring since the program doesn't run long enough to outweigh them.

4.4.3 Results

Lastly, we plot the absolute values useful for the analysis of the performance of the CPU algorithm with both vectorization and multi-threading via OpenMP.

First, the plot shown in image 11 seems to show approximately the same trend seen on the non-vectorized version (Figure 2), albeit with a much lower increase in average execution time between subsequent resolution values.

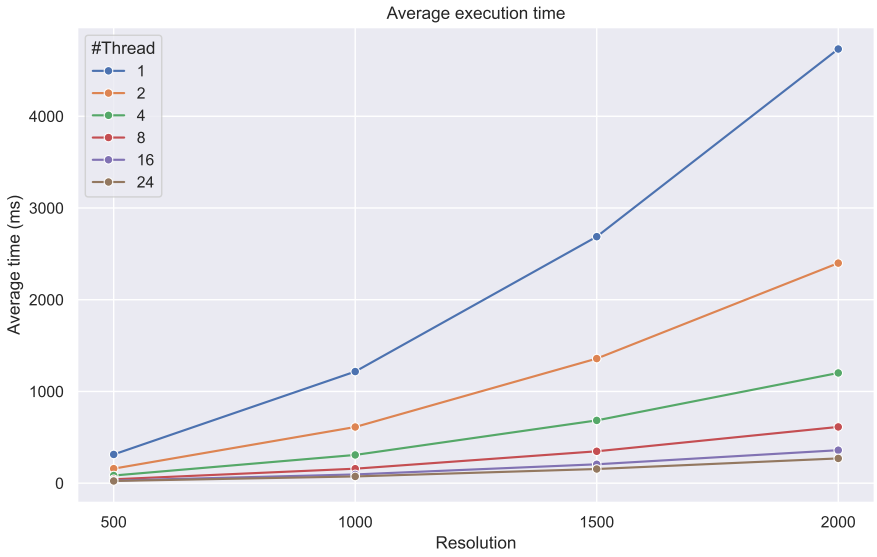


Figure 11: Average time with different number of threads and resolution

Further analysis on the speedup (Figure 12) and efficiency (Figure 13) plots highlight an almost perfect speedup and efficiency until 8 threads, and as the number of threads increases we can observe that the speedup is much closer to the linear trend we strive towards for the values of `#Thread` we analyzed, and thus the average efficiency only reaches as low as 0.75 on average.

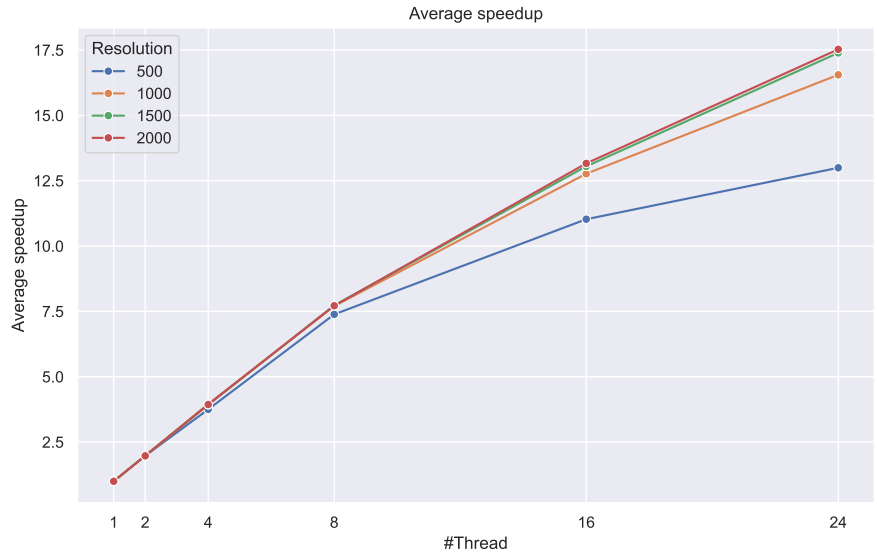


Figure 12: Average speedup with different number of threads and resolution

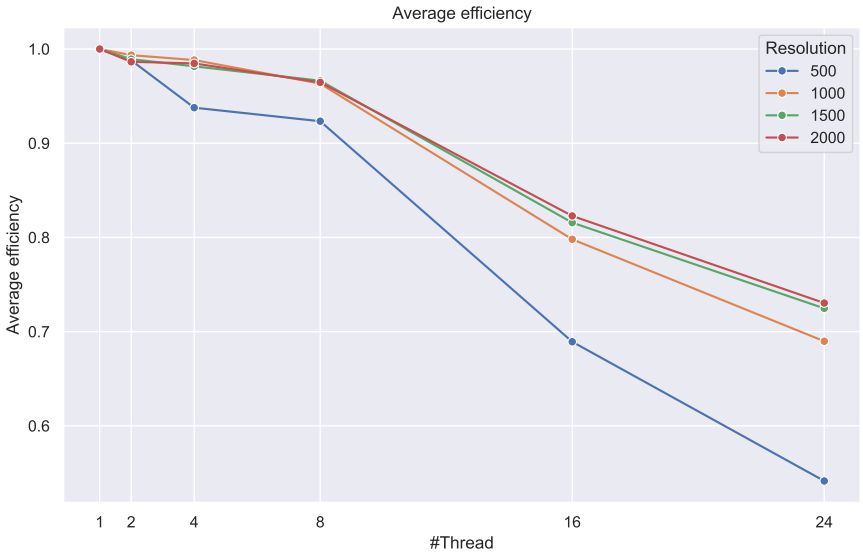


Figure 13: Average efficiency with different number of threads and resolution

Resolution	#Thread	Time (ms)	Speedup	Efficiency
500	1	313	1.00	1.00
500	2	159	1.98	0.99
500	4	84	3.75	0.94
500	8	42	7.39	0.92
500	16	28	11.03	0.69
500	24	24	13.00	0.54
1000	1	1217	1.00	1.00
1000	2	612	1.99	0.99
1000	4	308	3.95	0.99
1000	8	158	7.71	0.96
1000	16	95	12.77	0.80
1000	24	74	16.56	0.69
1500	1	2687	1.00	1.00
1500	2	1358	1.98	0.99
1500	4	684	3.93	0.98
1500	8	348	7.73	0.97
1500	16	206	13.05	0.82
1500	24	155	17.39	0.72
2000	1	4733	1.00	1.00
2000	2	2399	1.97	0.99
2000	4	1202	3.94	0.98
2000	8	613	7.72	0.96
2000	16	360	13.17	0.82
2000	24	270	17.53	0.73

Table 10: Execution time, speedup, and efficiency comparison by varying the resolution and the number of threads

5 GPU

The algorithm we used to generate the Mandelbrot set is easily transferable to a GPU kernel implementation. Since each point in the output image doesn't need any information from the neighboring ones. The calculations performed in each call to the kernel are simple arithmetic operations, this seems like the perfect algorithm to be run on the GPU.

Our GPU CUDA kernel implementation of the algorithm that generates the Mandelbrot set has few parameters, namely `TRHEAD_X`, `THREAD_Y`, `RESOLUTION`, and `__ftype`. The first two control the size of a GPU block, the third is inherited from the reference implementation, and the last is used to switch from `floats` and `doubles`.

5.1 Algorithm

We had to rewrite the reference algorithm in CUDA by creating a `kernel`. To do so, we had to remove the outer loop of the hotspot (See Listing 1) and re-write the `complex` type as two `__ftype` variables representing the real and the imaginary part respectively and by doing operations on those numbers in the same way as in the CPU vectorized code (see Section 4.3). We also applied the optimization outlined in 4.1.

```
__global__ void mandelbrot(int *const image) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col >= WIDTH || row >= HEIGHT) {
        return;
    }

    int pos = row * WIDTH + col;
    image[pos] = 0;
    __ftype c_re = col * STEP + MIN_X;
    __ftype c_im = row * STEP + MIN_Y;
    __ftype z_re = 0.0;
    __ftype z_im = 0.0;
    for (int i = 1; i <= ITERATIONS; i++)
    {
        __ftype z2_re = z_re * z_re - z_im * z_im;
        __ftype z2_im = 2.0 * z_re * z_im;

        // z = pow(z, 2) + c;
        z_re = z2_re + c_re;
        z_im = z2_im + c_im;

        // |z|2 = x2 + y2.
        __ftype abs2 = z_re * z_re + z_im * z_im;

        // If it is convergent
        if (abs2 >= 4)
        {
            image[pos] = i;
            return;
        }
    }
}
```

Listing 7: Mandelbrot CUDA kernel

At the call site of the kernel in the `main` we specify the number of threads per block by creating a tuple with the `X` and `Y` value of the block size and then computing the number of blocks as in the code snippet below.

```
    dim3 threadsPerBlock(THREADS_X, THREADS_Y);
    dim3 numBlocks(
        (WIDTH  + threadsPerBlock.x - 1) / threadsPerBlock.x,
        (HEIGHT + threadsPerBlock.y - 1) / threadsPerBlock.y
    );
```

5.2 `floats` vs `doubles`

We decided to analyze the performance and precision characteristics of our CUDA kernel implementation with both `float`s and `double`s in the same way we did in section 4.3.2 for the vectorized CPU implementation of the algorithm. Performance could be better with 32-bit floating point numbers since oftentimes GPU manufacturers pack more compute cores capable of FP32 calculations than the ones capable of FP64 computations in consumer-grade GPUs that are not strictly focused on precision work like the NVIDIA T400 2GB we're using for these experiments.

Table 11 confirms our suspicions and shows that on average the program that uses `float`s has a 20-25% lower execution time

Resolution	Float	Double
500	194	234
1000	565	742
1500	1187	1582
2000	2035	2734

Table 11: Execution time comparison between `float`s and `double`s. The program has been compiled with `THREAD_X=THREAD_Y=16`. Execution times are shown in ms.

In terms of precision 3% of the numbers in the final image differ between the program using `float`s and `double`s, and they once again lie on the edge of the set, as highlighted by Figure 14.

Table 12 shows more precisely the difference between the version using `float`s and `double`s.

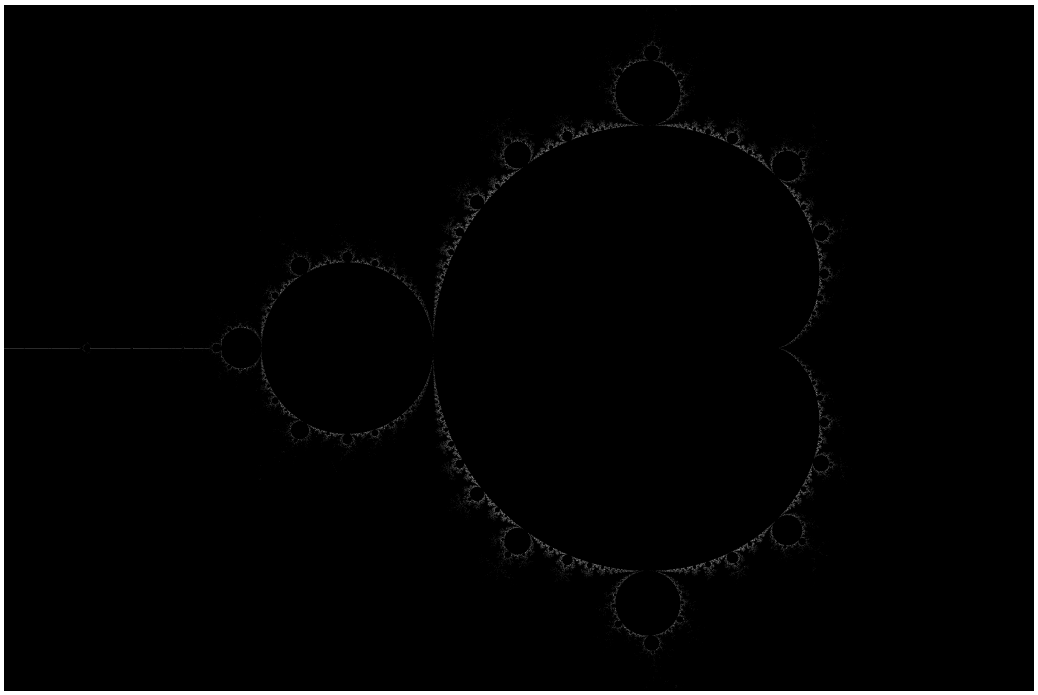


Figure 14: Difference between the images produced by using `float`s and `double`s. Produced with the outputs of programs running with `RESOLUTION=500`

Resolution	Points	%Difference
500	7531	3.01
1000	29683	2.97
1500	54006	2.40
2000	118413	2.96

Table 12: Number of different numbers in the final result between the program using `float`s and `double`s

It's also interesting to notice that the difference between the resulting images of 32-bit `floats` and 64-bit `doubles` calculated with the GPU kernel and the vectorized CPU function using FMA (table 5) instructions are quite similar. This may be caused by the optimizations applied by `nvc++`, which we used to compile our binaries for these experiments, meaning that the final binary could be using FMA instructions on the GPU as well. Nonetheless, there are still some incongruencies between the results in table 5 and 12, likely due to the differences in the precision of the hardware implementations of the instructions used by the two algorithms.

For the same reasons outlined in section 4.3.2, we decided to keep using `doubles` in the next experiments.

5.3 Results

Figure 15 shows that increasing the block size leads to better performance, almost up to the maximum we tested of a block size of 16×16 threads per block. In particular, with that configuration the hotspot calculation with `RESOLUTION=2000` was performed in little under 3 seconds, as shown in the raw data table (table 13, while the slowest configuration (block size 1×1) completed the same task in about 80 seconds.

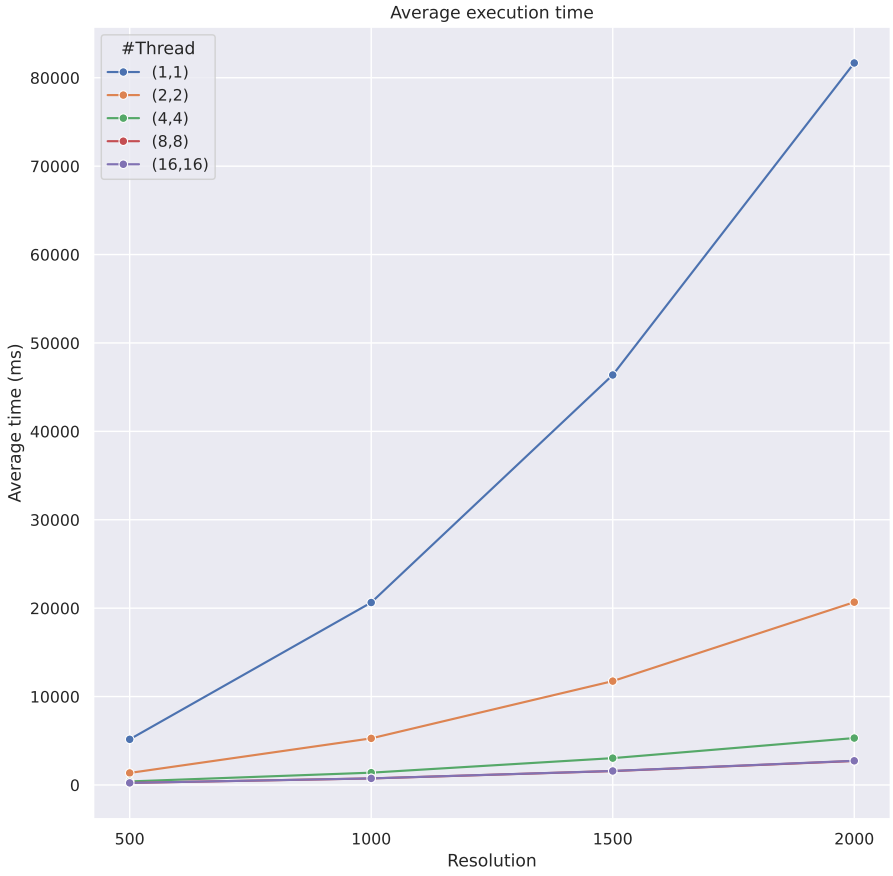


Figure 15: Average time per resolution using different combinations of threads

In general image 15, which only depicts the performance of blocks for which `THREAD_X=THREAD_Y`, shows that increasing the block size increases the performance of our kernel, with improvements decreasing as the block size increases, until stopping at blocks of size 8×8. The trends for blocks of size 8×8 and 16×16 overlap in the aforementioned Figure.

After a more thorough performance analysis of the kernel with varying block sizes shown in the heatmaps in Figure 16, we can see that the maximum performance is reached at around 32 threads per block, and all subsequent increases in block size bring little to no performance benefits. Square blocks with `THREAD_X=THREAD_Y=8`

are the first to reach the 32 threads per block threshold, and that's why that line overlaps with the 16×16 block size line in Figure 15. We were not able to understand why performance stopped increasing after that threshold.

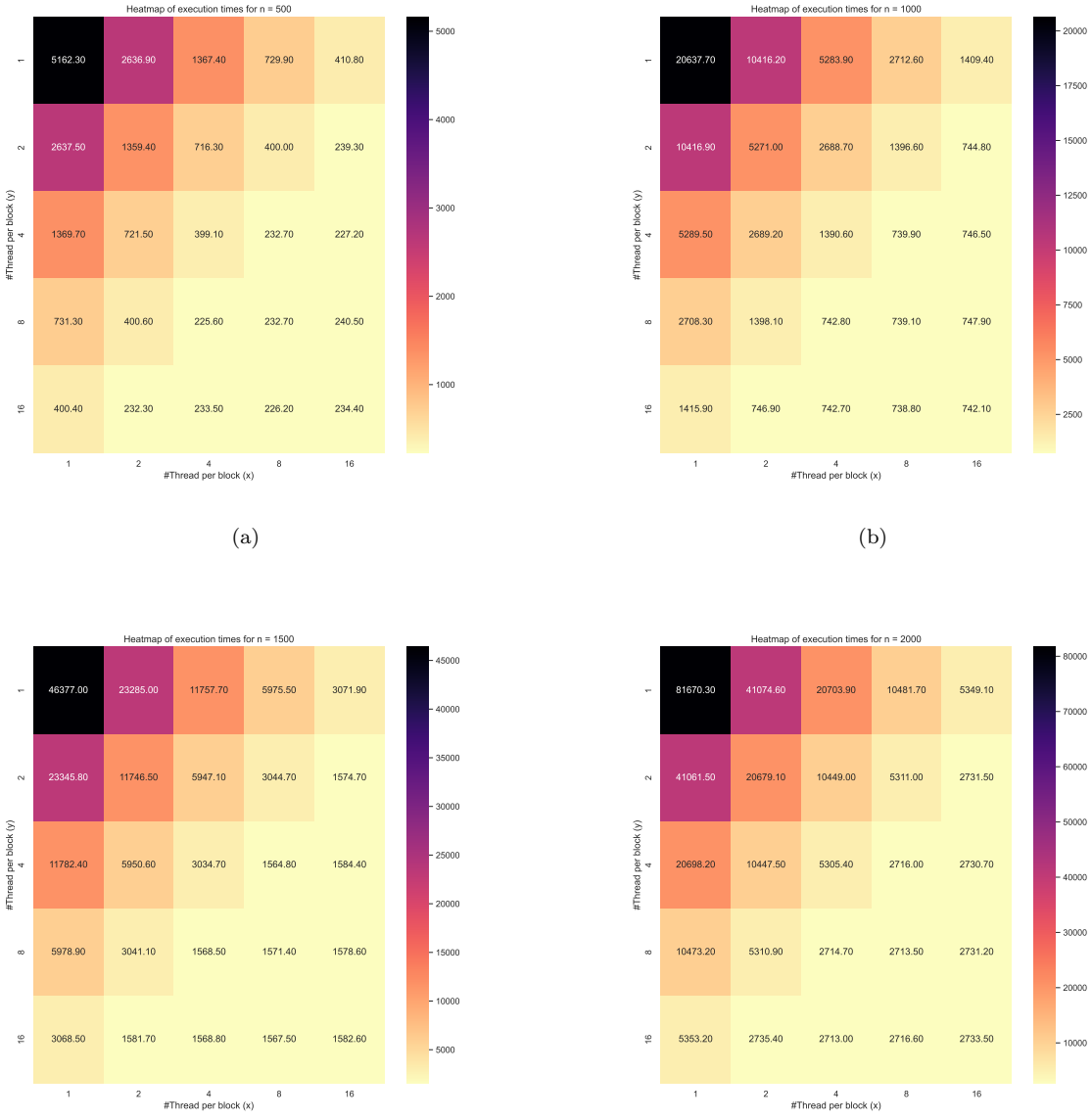


Figure 16: heatmaps of average execution times for different block sizes. Execution times are shown in ms.

Resolution	#Thread(x)	#Thread(y)	Time
500	1	1	5162
500	1	2	2637
500	1	4	1369
500	1	8	731
500	1	16	400
500	2	1	2636
500	2	2	1359
500	2	4	721
500	2	8	400
500	2	16	232
500	4	1	1367
500	4	2	716
500	4	4	399
500	4	8	225
500	4	16	233
500	8	1	729
500	8	2	400
500	8	4	232
500	8	8	232
500	8	16	226
500	16	1	410
500	16	2	239
500	16	4	227
500	16	8	240
500	16	16	234
1000	1	1	20637
1000	1	2	10416
1000	1	4	5289
1000	1	8	2708
1000	1	16	1415
1000	2	1	10416
1000	2	2	5271
1000	2	4	2689
1000	2	8	1398
1000	2	16	746
1000	4	1	5283
1000	4	2	2688
1000	4	4	1390
1000	4	8	742
1000	4	16	742
1000	8	1	2712
1000	8	2	1396
1000	8	4	739
1000	8	8	739
1000	8	16	738
1000	16	1	1409
1000	16	2	744
1000	16	4	746
1000	16	8	747
1000	16	16	742

Resolution	#Thread(x)	#Thread(y)	Time
1500	1	1	46377
1500	1	2	23345
1500	1	4	11782
1500	1	8	5978
1500	1	16	3068
1500	2	1	23285
1500	2	2	11746
1500	2	4	5950
1500	2	8	3041
1500	2	16	1581
1500	4	1	11757
1500	4	2	5947
1500	4	4	3034
1500	4	8	1568
1500	4	16	1568
1500	8	1	5975
1500	8	2	3044
1500	8	4	1564
1500	8	8	1571
1500	8	16	1567
1500	16	1	3071
1500	16	2	1574
1500	16	4	1584
1500	16	8	1578
1500	16	16	1582
2000	1	1	81670
2000	1	2	41061
2000	1	4	20698
2000	1	8	10473
2000	1	16	5353
2000	2	1	41074
2000	2	2	20679
2000	2	4	10447
2000	2	8	5310
2000	2	16	2735
2000	4	1	20703
2000	4	2	10449
2000	4	4	5305
2000	4	8	2714
2000	4	16	2713
2000	8	1	10481
2000	8	2	5311
2000	8	4	2716
2000	8	8	2713
2000	8	16	2716
2000	16	1	5349
2000	16	2	2731
2000	16	4	2730
2000	16	8	2731
2000	16	16	2733

Table 13: Execution times of the GPU algorithm with different block sizes. Execution times are shown in ms.

5.4 Comparison with CPU performance

From Figure 15 it's easy to notice that the execution time of even the most performant configuration of the GPU code is far slower than the equivalent CPU implementation.

Looking at the output of the NVIDIA Nsight profiling tool run on our binary (images 17 and 18) it's apparent that most of the execution time is spent in the call to `cudaMalloc` before invoking the CUDA kernel and after with the call of `cudaMemcpy` from GPU to RAM after the call to `mandelbrot`.

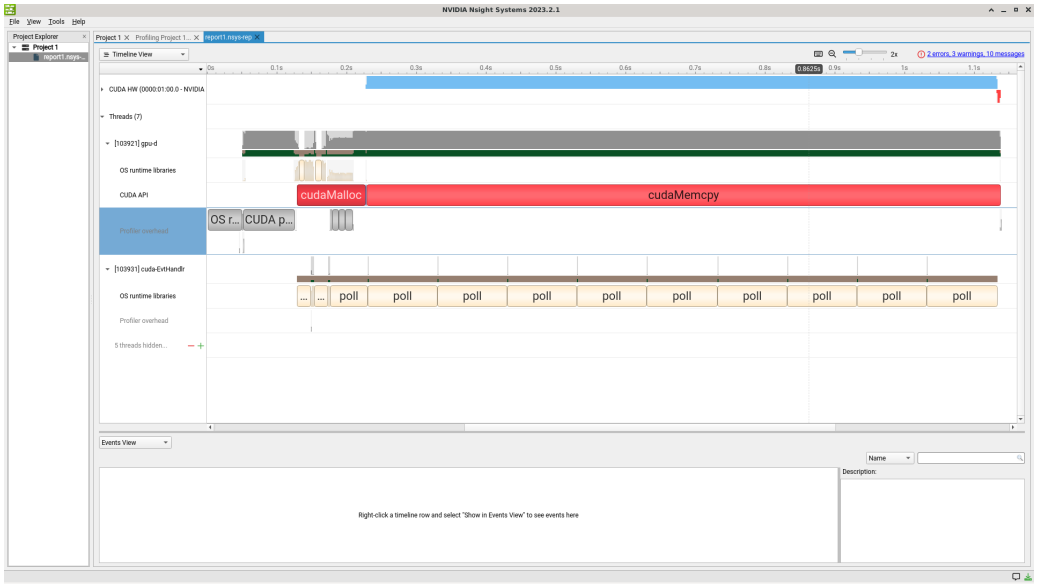


Figure 17: Screenshot of NVIDIA Nsight profiler. Profiled binary was compiled with `RESOLUTION=1000, FTYPE=double, THREAD_X=THREAD_Y=16`

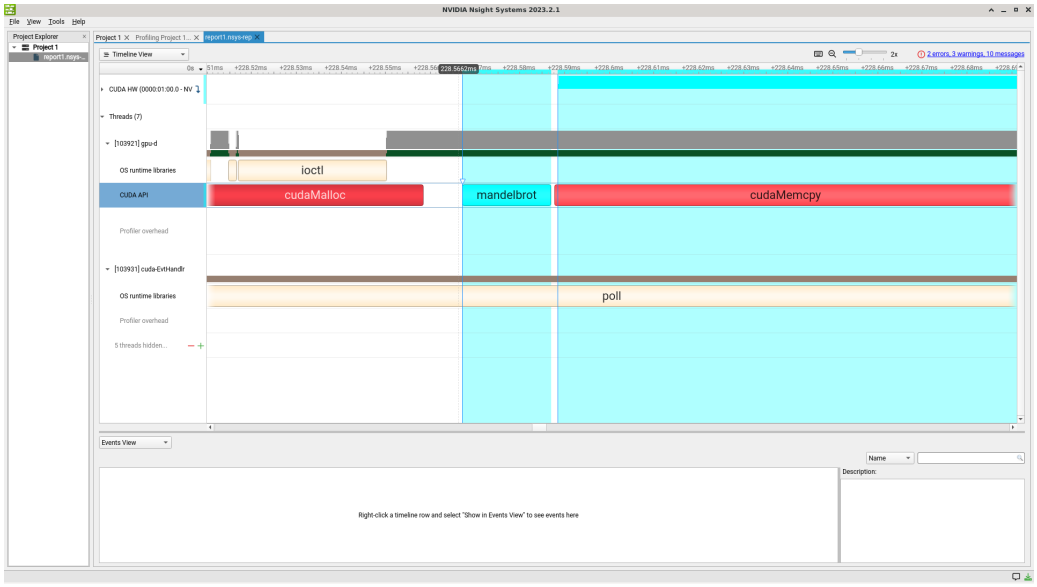


Figure 18: Screenshot of NVIDIA Nsight profiler zoomed on the main loop. Profiled binary was compiled with `RESOLUTION=1000, FTYPE=double, THREAD_X=THREAD_Y=16`

Analyzing the performance of our program more closely with the report produced by `nvprof` shown in listing 8, we can see that the call to `cudaMemcpy` alone takes the overwhelming majority of the execution time of the program, at about 91%, followed by 8.5% on the call to `cudaMalloc`. It's important to note that the call to `cudaMemcpy` also includes an implicit call to `cudaEventSynchronize`, which means that the actual time spent copying the memory from the GPU to RAM is much lower, at approximately 4.4ms, as shown in the `GPU activities` section in the aforementioned output.

```
==45688== NVPROF is profiling process 45688, command: ../bin/mandelbrot_gpu_1000_double
Time elapsed: 887 ms.
==45688== Profiling application: ../bin/mandelbrot_gpu_1000_double
==45688== Profiling result:
        Type    Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:
        99.34%  662.55ms         1  662.55ms  662.55ms  662.55ms  mandelbrot(int*)
        0.66%   4.3873ms         1  4.3873ms  4.3873ms  4.3873ms  [CUDA memcpy DtoH]
API calls:
        91.38%  667.00ms         1  667.00ms  667.00ms  667.00ms  cudaMemcpy
        8.55%   62.418ms         1  62.418ms  62.418ms  62.418ms  cudaMalloc
        0.02%   171.56us       101  1.6980us     206ns  71.205us  cuDeviceGetAttribute
        0.02%   152.00us         1  152.00us  152.00us  152.00us  cuDeviceTotalMem
        0.01%   93.744us         1  93.744us  93.744us  93.744us  cudaFree
        0.00%   36.024us         1  36.024us  36.024us  36.024us  cuDeviceGetName
        0.00%   15.066us         1  15.066us  15.066us  15.066us  cudaLaunchKernel
        0.00%   4.1770us         1  4.1770us  4.1770us  4.1770us  cuDeviceGetPCIBusId
        0.00%   2.4880us         3     829ns     244ns  1.7630us  cuDeviceGetCount
        0.00%   1.2830us         2     641ns     221ns  1.0620us  cuDeviceGet
        0.00%      468ns         1     468ns     468ns     468ns  cuDeviceGetUuid
```

Listing 8: `nvprof` output. Profiled binary was compiled with `RESOLUTION=1000, FTYPE=double, THREAD_X=THREAD_Y=16`

Furthermore, Table 14 shows the amount of time taken by the full program, the kernel, and some of the heaviest CUDA API calls. While the percentage of time taken by these calls decreases as resolution increases, the CUDA API function calls still take a considerable amount of time.

Resolution	Program	Kernel	`cudaMemcpy`	`cudaMalloc`
500	732	453	1.4	89
1000	887	662	4.4	62
1500	1721	1476	9.9	53
2000	2910	2608	16.7	56

Table 14: Execution times of the whole program, and CUDA APU function calls. Extracted from a single run with `nvprof` for each `RESOLUTION` value. Execution times are shown in ms.