Report Mandelbrot

Authors:
Riccardo Isola - S4943369
Gabriele Dellepere - S4944557
Kevin Cattaneo - S4944382

Index

Head of analysis

Algorithm: the code provided has quadratic N^3 (cubical) complexity.
Tools: we used the *icpx* compiler, the Intel one, and Intel Advisor GUI to perform the most of the analysis.
Machine: the machine on which we run the code has 20 processors:

- 12 core
  - 8 performance cores with hyperthreading up to 2
  - 4 efficiency cores

For the current analysis we decided to fix the RESOLUTION that is WIDTHxHEIGHT and to iterate different runs over different ITERATIONS.
We also changed the original time function with the OpenMP library one to be more precise (the original one was rounding on seconds).

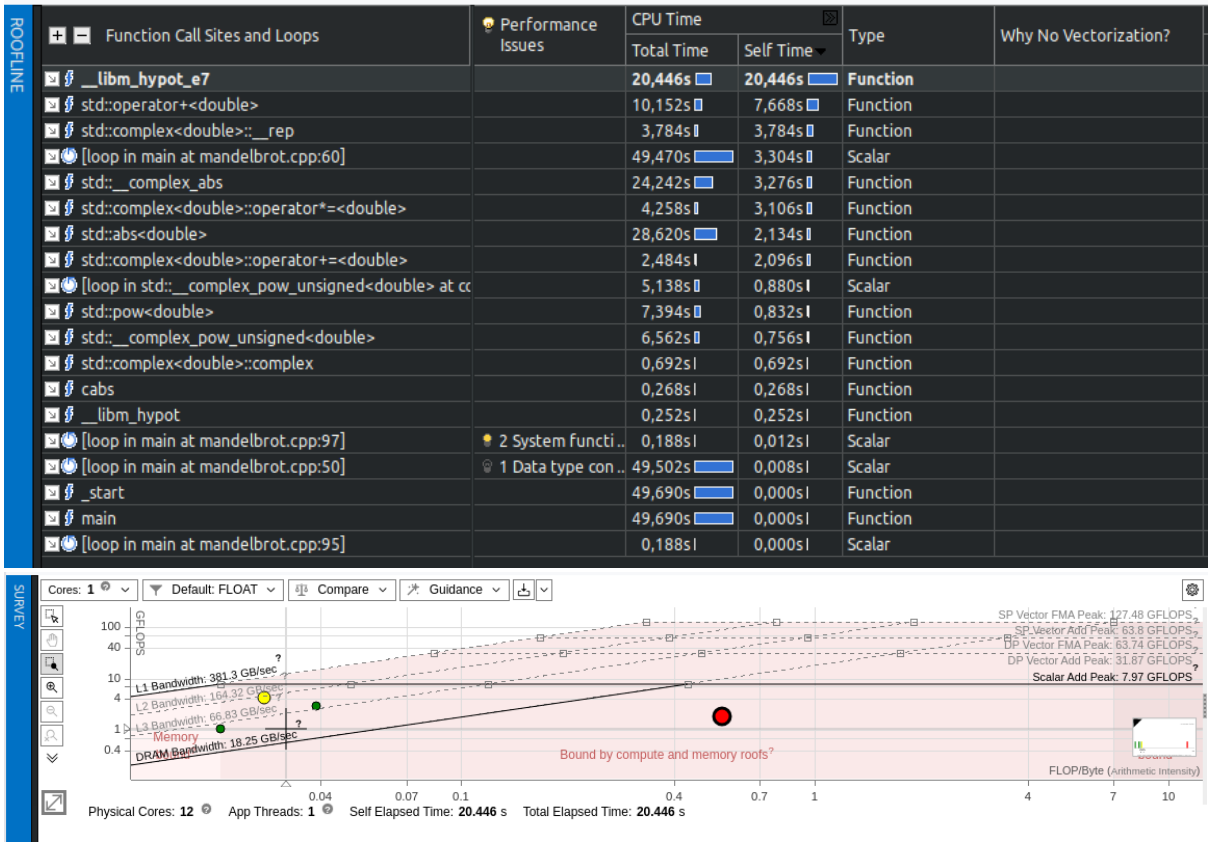
Hotspot identification

In first place we compiled the program:

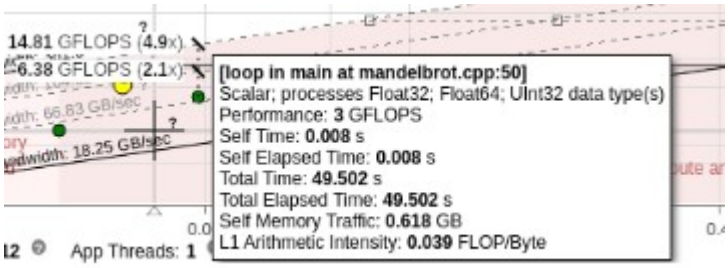
- **`compiling line`**: `icpx -g -fopenmp mandelbrot.cpp`
- **`data size`**: `RESOLUTION=1000, ITERATIONS=700`
- **`time taken`**: `49.70 seconds`

Then we put the executable into intel advisor to perform a detailed analysis, where we identified the following hotspots.
What we discovered is that the hotspot is the main loop at line 44, contained in the one at line 34, so the one that performs the mandelbrot computation. So the main objective will be trying to parallelize the external one, since the inner one has the dependence on the previous value of z at each iteration.

Name	Time taken (seconds)
Overall program	49.70
_libm_hypot	20.45
operator+<double>	7.67
complex<double>	3.79
loop in main - line 60	3.78



As we can see from the survey section, the main hotspots are the math operations of _libm_hypot that performs the Euclidean distance, the override of the operator + for double in the library std, the conversion into complex from double, then the main loop that we can optimize, that performs the power operation. The remaining most taken times are taken again by other math operations like abs on double and operator + on complex type.
As we can see on the roofline, the point that we can optimize is the third from the left, that is the green one representing the main loop.



Vectorization issues

To obtain the report about vectorization infos, we specified the level 3:

First
- **compiling line**: `icpx -O3 -xHost -qopt-report=3 mandelbrot.cpp`
- **data size**: `RESOLUTION=1000, ITERATIONS=700`
- **time taken**: `2.27 seconds`

The mandelbrot.optrpt report told us the following:

- LOOP BEGIN at mandelbrot.cpp (50, 5)
    - **remark** #15541: loop was not vectorized: outer loop is not an auto-vectorization candidate.

- LOOP BEGIN at mandelbrot.cpp (60, 9)
    - **remark** #15344: Loop was not vectorized: vector dependence prevents vectorization
    - that is the dependence on the z we pointed out previously

The reason we cannot apply vectorization to the inner loop is that each iteration depends on the value of z produced by the previous one (to vectorize it we should already know beforehand all values assumed by z, which, besides being impossible, would actually make the whole loop pointless).

Apparently, we cannot vectorize the outer loop either, even though it looks like it could be possible by “simply” placing the *z* and *c* variables for different values of *pos* inside a vector. After long considerations we ended up with the following conclusions:

- First and foremost, the break statement inside the inner loop is problematic: for some *pos*, *z* causes the break earlier than others, this could be solved by avoiding the break altogether and setting the update to *image[pos]* as a conditional assignment controlled by a ternary operator (which is vectorizable). This solution would however cause some overhead because of the addition of several useless iterations
- Secondly, *z* and *c* are complex numbers, which means to perform the pow operation the processor needs to:
    - convert z into polar coordinates
    - apply exponentiation to the modulus and multiplication to the angle
    - re-convert to cartesian coordinates (to allow the sum with c)

  Each of these steps comes with some problems. To convert the number into polar coordinates we need the atan function, which could be approximated with a Taylor series, but it is not implemented by default as a vectorized operation. The same problem is posed by the conversion back to cartesian coordinates (we can approximate sin and cos with Taylor but there is no default vectorized implementation). Lastly, even the simple exponentiation applied to the modulus is not vectorizable by default. It is worth to notice that the intel compiler provides specific implementations for some of these methods, however using them would make the code less portable and give us little to no benefit (because of the overhead still caused by the break removal), meaning that for this project, we’re going to give up on vectorization.

Best sequential time

Now we perform different runs to find the best sequential time, to be used as reference for the further parallel application, by passing several arguments and flags to the intel compiler.

Original without optimizations
- **compiling line**: `icpx -O0 -fopenmp mandelbrot.cpp`
- **data size**: `RESOLUTION=1000, ITERATIONS=700`
- **time taken**: `50.488 seconds`

Compared to the one with the flag optimizations:

- **compiling line**: icpx -g -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=700
- **time taken**: 2.422 seconds

Then we tried different combinations of flags to find the best combination in terms of time, to make this difference more visible, we also increased the data size. With that we identified the best sequential case with data size equal to 10000 iterations.
- **compiling line**: icpx -fopenmp -O3 -xHost mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 31.3712 seconds

- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 31.2874 seconds

- **compiling line**: icpx -fopenmp -O3 -xHost -ipo mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 31.4665 seconds

- **compiling line**: icpx -fopenmp -O3 -xHost -ipo -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 31.3204 seconds

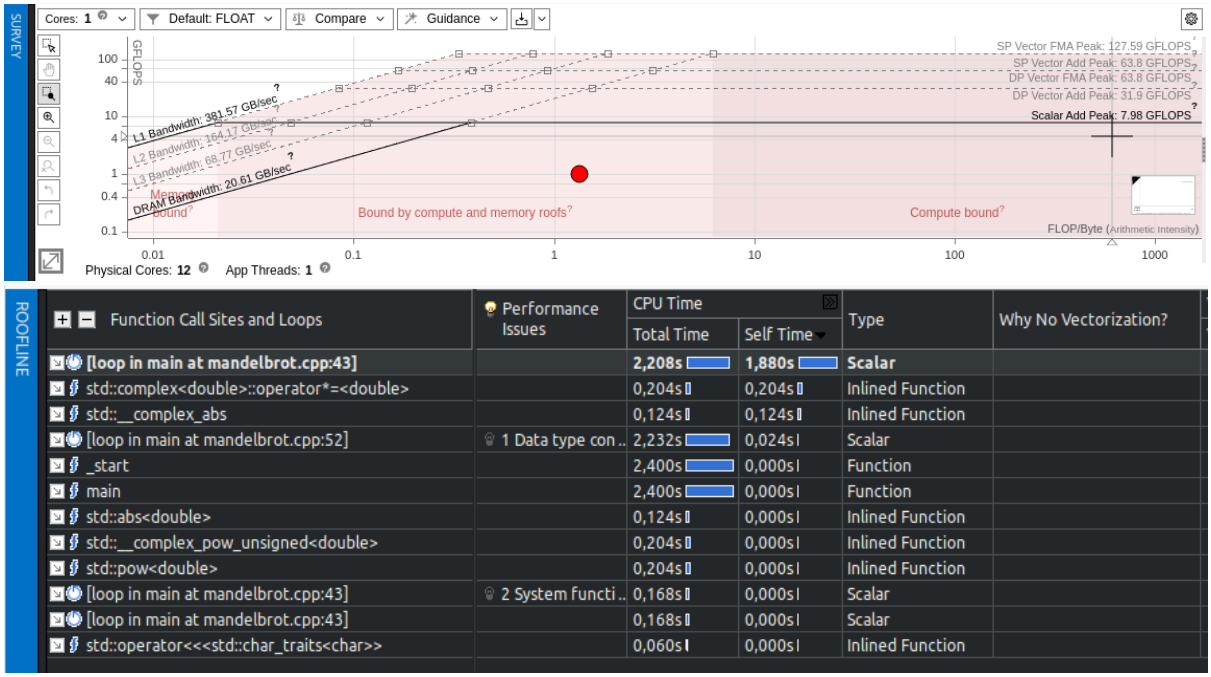
We see a little difference, so there is no best combination.

Overall we are applying the best optimizations arguments:

- **O3**: to allow the compiler to optimize the code, so that it can reorder the operations in the most efficient way
- **xHost**: to achieve the best assembly instructions specifically on the current machine architecture
- **ipo**: to enable interprocedural optimizations; we observed that with this flag the program didn't achieve better performances
- **fast-math**: to reduce the time needed for mathematical operations, although reducing the precision of our calculations.

We also check on the roofline the result obtained so far with the combination of flags, also decreasing the data size for a matter of time spent by Intel Advisor:
- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp

- **data size**: RESOLUTION=1000, ITERATIONS=700
- **time taken**: 2.23492 seconds



By exploiting the optimization flags we achieved the reduction in time of the hotspots, so the math computations on double and complex numbers.
Also the advisor suggests that we have some data type conversion between two different widths, that's because of the conversion from complex to real and vice versa, that can't be avoided because of the algorithm implementation.
Also there are some notifications about 2 system calls used and also a reduction suggested, but those can't apply actually to the algorithm, so it's an hallucination of Advisor.

OpenMP Parallelization

To compare we fix the same data size of before and we run the parallelization.
On line 49 we put the following line:
 # pragma omp parallel for default(none) shared(image)

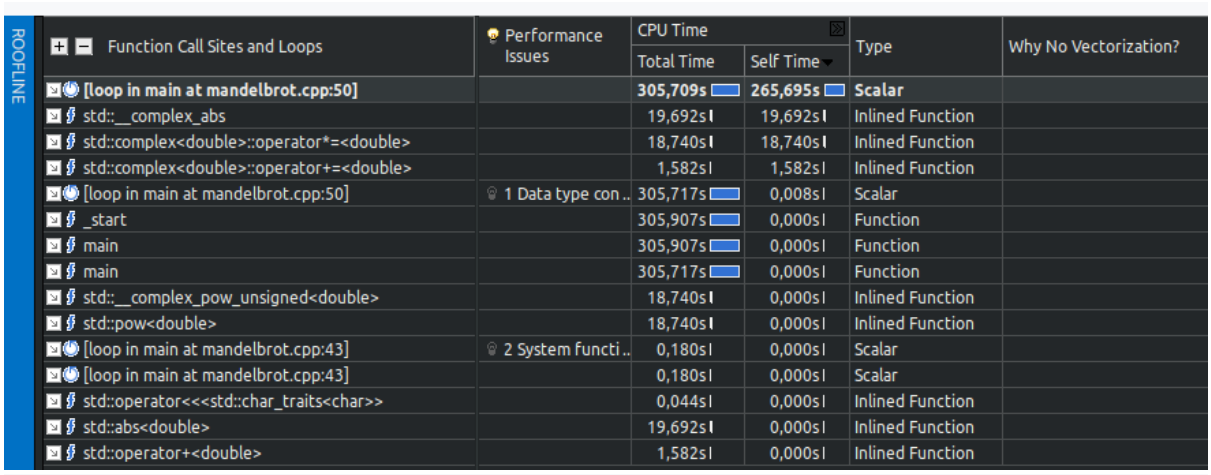
We compare the original best sequential run without pragma:
- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000

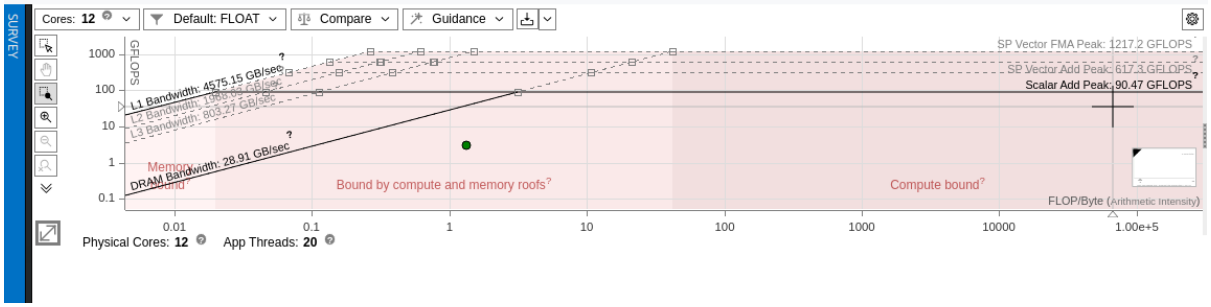
- **time taken**: 31.2413 seconds

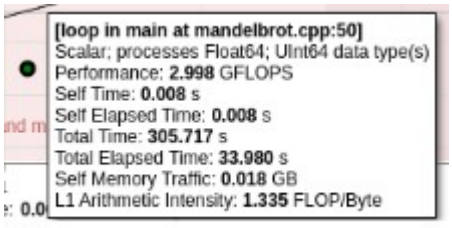
With pragma enabled:
- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 4.10494 seconds

We reduced the time needed by 8 times, also exploiting the maximum power of our machine in terms of threads.
Now we increase the data size to test the limits by trying different combinations of the usage of the *pragma* instructions.

- **compiling line**: icpx -fopenmp -O3 -xHost -ipo -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=80000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 30.3726 seconds







We increased our data size and our arithmetic intensity achieving great results with respect to the non-parallel one. For some reason Advisor hallucinates and gives some 300s of runtime even if the program is correctly recognized to have ~30s of runtime.

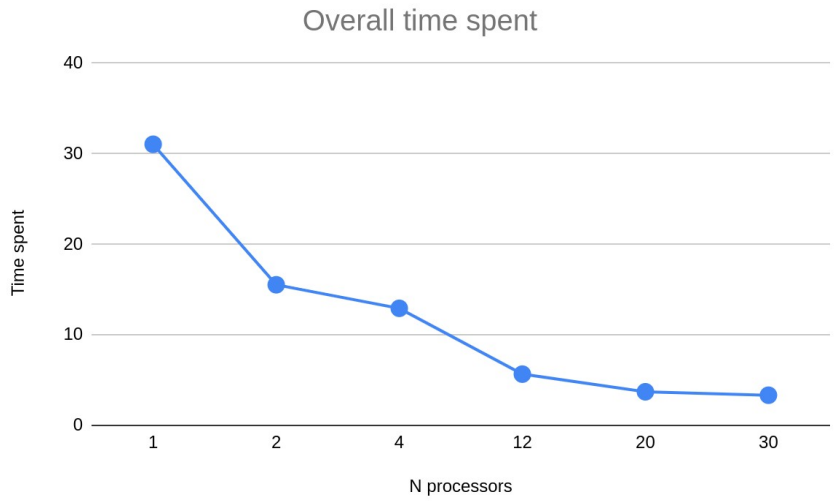
Speedup and efficiency

As possible number of processors, we chose:

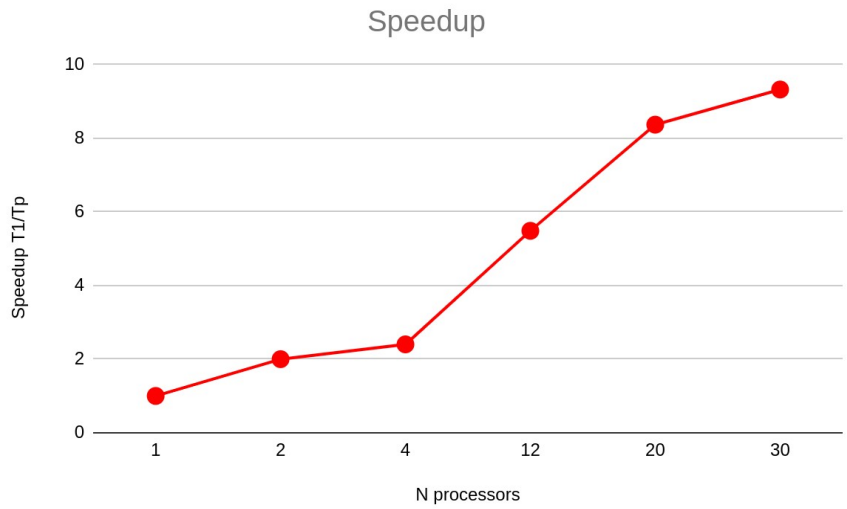
- 1: sequential run
- 2: enable parallelization
- 4: equal to the number of performance cores
- 12: equal to the number of all cores without considering hyperthreading
- 20: the maximum value of processors considering hyperthreading up to 2
- 30: to observe any overhead

`compiling line``: icpx -g -fopenmp -O3 -xHost -ipo -ffast-math mandelbrot.cpp`

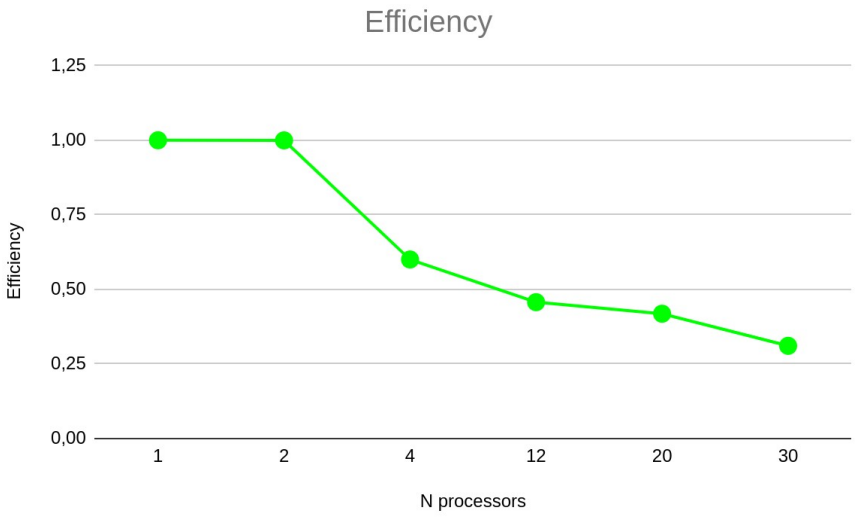
Data size	`RESOLUTION=1000`	`ITERATIONS=10000`	
N processors	Time spent	Speedup T1/Tp	Efficiency
1	31,05	1	1.00
2	15,53	2.00	1.00
4	12,93	2.40	0.80
12	5,66	5.48	0.46
20	3,71	8.37	0.42
30	3,33	9.32	0.31



We observe that after enabling the parallelization with 2 processors, the time spent drastically decreases, while remaining more or less stable after 12 processors.



Here we see that the speedup increases in different sections, in particular after enabling the parallelization with 2 processors, then a huge peak after 4.



On the other hand, the efficiency remains more or less stable up to 4 processors, then starts decreasing, but the most decreased distance is from 2 processors.

CUDA Parallelization

As first thing we explored some specifications of the machine on which we are running on via *devicequery:*

```
Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          12.2 / 12.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15102 MBytes
(15835660288 bytes)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072),
2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048
layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768),
2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
65535)
[...]
```

So the maximum number of threads that we can use per block is 1024, scheduled in warps of 32 threads. Also we have available 40 Streaming Multiprocessors (each with 2 warp schedulers), for a total available maximum of 2560 concurrent threads scheduled at each clock cycle.

The main idea was to unroll the for loop inside the function *main* that generates the mandelbrot matrix via indexes, and so we parallelize each pos, so pixels.
So our kernel is implemented as follows:

```
__global__ void kernel(int* image)
{
 int pos = threadIdx.x + blockIdx.x * blockDim.x;
```

```
 if(pos > WIDTH*HEIGHT) return;

 // evaluate derivatives

 image[pos] = 0;

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

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

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

}
```

And in the main:

```
const int size=WIDTH*HEIGHT;

   int *image_dev;

   cudaMalloc((void**) &image_dev, size);

   cudaMemcpy(image_dev, image, size, cudaMemcpyHostToDevice);

   dim3 block_size(BLOCK_SIZE);
   dim3 grid_size = dim3((size - 1) / BLOCK_SIZE + 1);

   // Execute the modified version using same data
   kernel<<<grid_size, block_size>>>(image_dev);
   cudaMemcpy(image, image_dev, size, cudaMemcpyDeviceToHost);
```

```
    cudaDeviceSynchronize();

    cudaFree(image_dev);
```

The original run, in the remote Colab machine:
- **compiling line**: g++ -O3 -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 73.682 seconds

The run with the parallelization:
- **compiling line**: nvcc mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **time taken**: 5.32149 seconds
- **note:** with a BLOCK_SIZE of 256 (we obtain similar result also with 32 since we schedule the entire warp), in this case to exploit multiple scheduling of the warps

In the lab machine with OpenMP we obtained:
- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=1000, ITERATIONS=10000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 4.10494 seconds

It seems that at this time the computation with the accelerator didn't win over the OpenMP implementation, even if for just a second. On the other hand we expect that if we increase the resolution we increase the number of same operations that need to be applied just on a larger scale (matrix). Given that the accelerator should perform better than the OpenMP implementation, since we scaled on the iterations.

The run with the parallelization:
- **compiling line**: nvcc mandelbrot.cpp
- **data size**: RESOLUTION=3000, ITERATIONS=5000
- **time taken**: 11.0791 seconds
- **note:** with a BLOCK_SIZE of 256

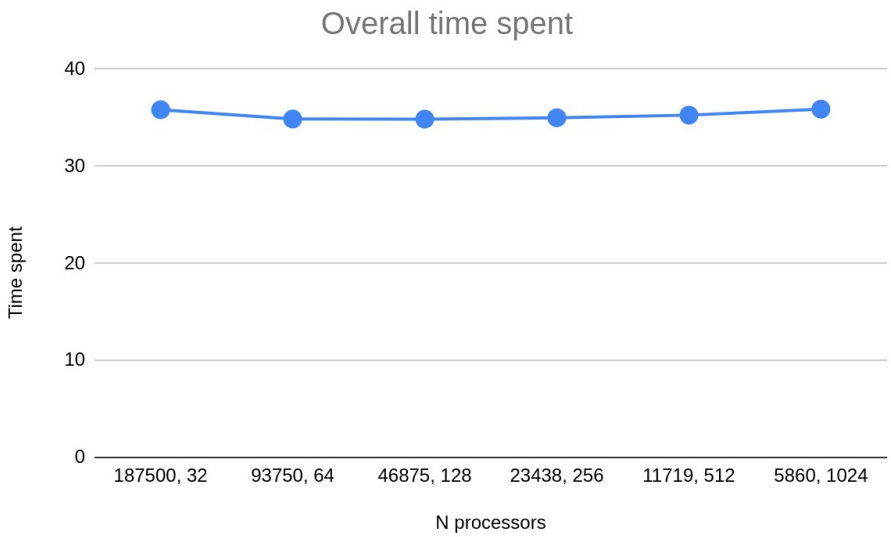
In the lab machine with OpenMP we obtained:
- **compiling line**: icpx -fopenmp -O3 -xHost -ffast-math mandelbrot.cpp
- **data size**: RESOLUTION=3000, ITERATIONS=5000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 20.171 seconds

As expected we obtained a better result from the GPU when scaling over resolutions.

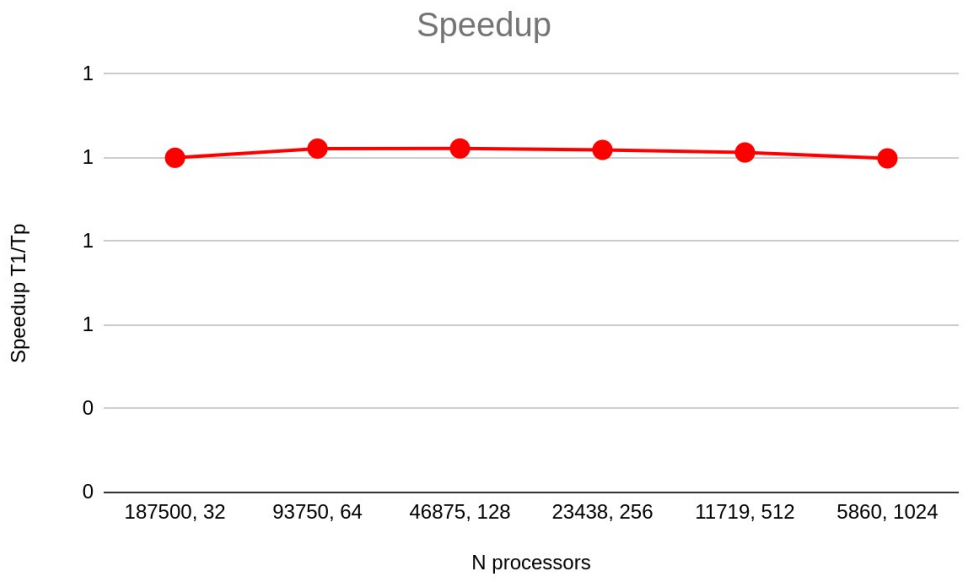
Speedup with accelerator

For this computation we used different combinations of block size and by doing so also of grid size, that is computed starting from the block width and height, observing the time spent. We start from a block_size that is above the warp size of 32 threads. Respectively in the following <<<grid_size, block_size>>>

Data size	RESOLUTION=1000	ITERATIONS=70000
N processors	Time spent	Speedup T1/Tp
187500, 32	35,83	1
93750, 64	34,87	1
46875, 128	34,86	1
23438, 256	35,00	1
11719, 512	35,27	1
5860, 1024	35,89	1



Since the gridSize and the blockSize are inverse proportional dependent, the number of threads scheduled is the same. The time achieved is almost the same in each run.



The same applies also for speedup, not achieving great difference.

MPI Parallelization

The last form of parallelization we could exploit was to split the computation of different sections of the image among different processes and use message passing to rebuild the final mandelbrot image inside the root process. To achieve this, we used MPI: the idea was simple, split the workload by assigning to each process an “equal” amount of pixels to fill, run the computations independently, and then perform a Gather on the partial results to assemble the final image. The only caveat of this procedure was that there was no guarantee that the amount of pixels in the image would be multiple of the number of processes: this tiny detail meant that we had to use the MPI_Gatherv instead of the simpler MPI_Gather (each process could be responsible of either n or n+1 pixels). We also had to define 2 auxiliary functions, one to decide how much work to assign to each process and another one to compute the array displacements to use in the Gatherv.

```
// returns the array with all work assignments
inline int* splitWorkEqually(const int id, const int processes, const int totalWorkSize, int *workStart, int *workSize) {

    int avgWork = totalWorkSize / processes;
    int remainder = totalWorkSize % processes;

    int *assignments = new int[processes];

    for (int j = 0; j < remainder; ++j) assignments[j] = avgWork + 1;

    for (int j = remainder; j < processes; ++j) assignments[j] = avgWork;

    *workStart = id * avgWork + min(id, remainder);
    *workSize = assignments[id];

    return assignments;
}

// needed for the displacements
inline int* getPrefixSum(const int *array, int arraySize) {

    int *prefixSum = new int[arraySize];
    prefixSum[0] = 0;

    for(int j = 1; j < arraySize; ++j) {
        prefixSum[j] = prefixSum[j-1] + array[j-1];
    }

    return prefixSum;
}
```

Completed this part it was simply a question of adding the MPI instructions in the main to initialize MPI, retrieve the communicator rank and size, set the workstart and worksize depending on the rank and then perform the gatherv and finalize.

```
int* image = nullptr;
int* imageBuffer = nullptr;

double start;

#ifdef _MPI_

    int commSize;
    int* assignments;
    int* displacements;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &commSize);

#endif

if (id == 0) {

    start = omp_get_wtime();
    image = new int[HEIGHT * WIDTH];
    imageBuffer = image; // this is important so that we can use the same code both with and without _MPI_
}

#ifdef _MPI_

    // assignments and displacements are only needed by root process, but since they do not occupy
    // much space I will free them only after the computation has completed
    assignments = splitWorkEqually(id, commSize, HEIGHT * WIDTH, &workStart, &workSize);
    displacements = getPrefixSum(assignments, commSize);

    imageBuffer = new int[workSize];

#endif
```

```
#ifdef _MPI_

    MPI_Gatherv(imageBuffer, workSize, MPI_INT, image, assignments, displacements, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();

#endif
```

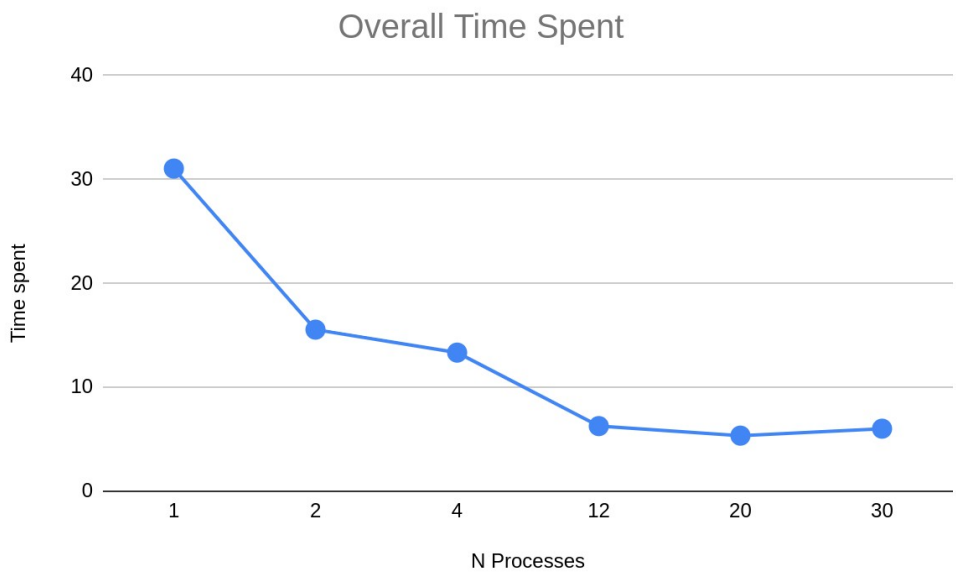
MPI vs OpenMP speedup

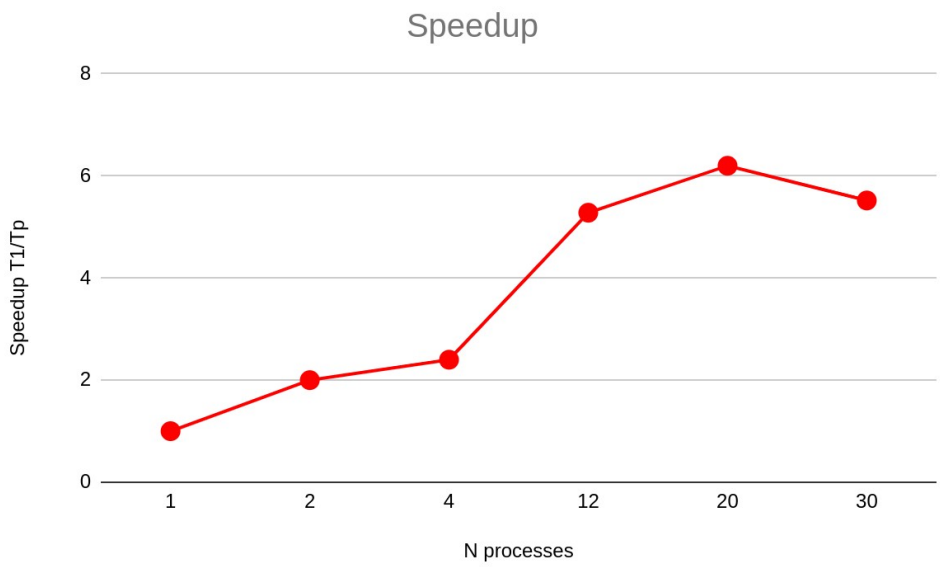
We first tried to use just MPI and reset the iterations amount:

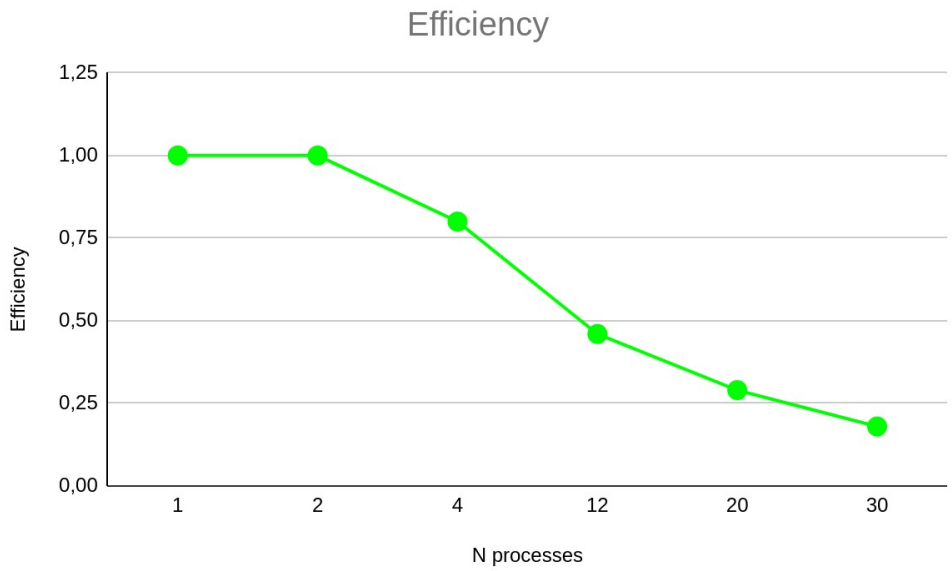
- **compiling line**: `mpiicpx -fopenmp -O3 -xHost -ffast-math -D_MPI_ mandelbot.cpp`
- **exec line:** `mpiexec -np <N processes> ./a.out image`

`Note: at this point we had wrapped the OpenMP pragma inside an #ifdef OMP, the -fopenmp parameter is there just to avoid compilation errors because of the function used to capture the time`

Data size	RESOLUTION=1000	ITERATIONS=10000	
N processes	Time spent	Speedup T1/Tp	Efficiency
1	31.05	1	1.00
2	15.55	2.00	1.00
4	12.94	2.40	0.80
12	5.87	5.28	0,46
20	5.35	5.80	0.29
30	5.62	5.52	0.18


Overall Time Spent
Time spent
40
30
20
10
0
1
2
4
12
20
30
N Processes


Speedup
Speedup T1/Tp
8
6
4
2
0
1
2
4
12
20
30
N processes



We can see that with 1 process there is no difference between this version of the program and the sequential one, as expected. Splitting the computation among many processes, instead, gives us a speedup similar to the one we obtained with OpenMP: we must acknowledge, however, that using MPI on a single machine is slightly slower than just OpenMP, since the message passing causes overhead. We also need to consider that context switch between processes is way slower with respect to threads of the same process, meaning that if we use all 20 available cores to compute the mandelbrot on 20 different process, and at the same time our machine is running some other service (just like any normal machine), we should expect a lot of heavy context switches to happen during the execution of our program. This became even more clear when we increased the number of iterations to the amount that took the OpenMP version 30 seconds to complete:

- **compiling line**: `mpiicpx -fopenmp -O3 -xHost -ffast-math -DITERATIONS=80000 -D_MPI_ mandelbot.cpp`
- **exec line:** `mpiexec -np 20 ./a.out image`
- **data size**: `RESOLUTION=1000, ITERATIONS=80000`
- **time taken**: `48.4521 seconds`

After these tests, we wondered what would happen if we tried to combine MPI with OpenMP: it's no wonder that it would make the program very efficient if we could split it across multiple machines, but would it still be convenient on a single machine or would it be better to just stick to OpenMP in that case?
Hence we run further tests with both MPI and OpenMP enabled:

- **compiling line**: `mpiicpx -fopenmp -O3 -xHost -ffast-math -DITERATIONS=80000 -DOMP -D_MPI_ mandelbot.cpp`
- **exec line:** `mpiexec -np 1 ./a.out image`

- **data size**: RESOLUTION=1000, ITERATIONS=80000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 34.5305 seconds

- **compiling line**: mpiicpx -fopenmp -O3 -xHost -ffast-math -DITERATIONS=80000 -DOMP -D_MPI_ mandelbot.cpp
- **exec line:** mpiexec -np 2 ./a.out image
- **data size**: RESOLUTION=1000, ITERATIONS=80000
- **number of threads**: 20 (decided by scheduler)
- **time taken**: 21.7537 seconds

We could see that there was some form of speedup, however it was most likely caused by the CPU exploiting the hyperthreading to hide some latencies.

Data size	RESOLUTION=1000	ITERATIONS=80000	
N processes	Time spent	Speedup T1/Tp	Efficiency
1	34.53	1	1,00
2	21.75	1.59	0.79
4	20.54	1.68	0,42
8	19.53	1.77	0.22
20	19.72	1.75	0.09

When we tried to increase the number of processes we basically saw no further improvement.

Unfortunately, there was no way to test the efficiency of the combination of MPI with CUDA, even if our implementation supports using both at the same time, because there's no point in doing that on a single machine (which is equipped with one GPU).

Conclusions

We observed that, in the best sequential case, our code performs the run in about 30 seconds, on an amount of data that from the original unoptimized sequential run is 8 times more.
By applying the *pragma* instructions to achieve better performance exploiting our processors via the OpenMP library, we observed an improvement that follows the increase of the number of threads. By just using 2 processors, we halved the time run, and we halved again at 12 processors usage.
The optimal number of threads to be used, without losing too much in terms of efficiency is 12. We can get better performance by increasing that number to the

maximum number of processors, that is 20, but the gain is of just a few seconds, not so relevant.
Finally, by applying a number of processors greater than the one owned in the machine, we don’t observe any major overhead that significantly delays the time run of our code.

Regarding the accelerator we saw the difference from the OpenMP version when scaling over the resolution, that is the applying the same operation on a larger scale. Also since we have the same amount of threads computed with the difference block size and grid size, we didn’t achieve significant differences in terms of speedup on several combinations of <<<gridSize, blockSize>>>.

When it comes to MPI, we saw that on a single machine it has similar behavior to OpenMP, with a slight overhead caused by the message passing and the heavier context switches. Even in this case, if we are fine with ~50% efficiency, the optimal number of processes to launch is 12. We saw how using MPI and OpenMP on the same machine can provide a small additional speedup, but it is clear that this is not the intended usage of the 2 libraries: there is probably no point in using MPI when all we have available is one single device, as OpenMP does the same thing with less overhead and with higher level instructions. We did not have the cluster available, so unfortunately in this report we are missing results representing the true potential of both libraries (just like for MPI+CUDA).