Report Mandelbrot OpenMP

**Authors:**

*Riccardo Isola - S4943369*

*Gabriele Dellepere - S4944557*

*Kevin Cattaneo - S4944382*

# Index

[**Index 1**](#_heading=h.30j0zll)

[**Head of analysis 1**](#_heading=h.1fob9te)

[**Hotspot identification 1**](#_heading=h.3znysh7)

[First optimization (manual) 3](#_heading=h.2et92p0)

[**Vectorization issues 3**](#_heading=h.tyjcwt)

[**Best sequential time 4**](#_heading=h.3dy6vkm)

[**OpenMP Parallelization 5**](#_heading=)

[**Speedup and efficiency 7**](#_heading=)

[**Conclusions 9**](#_heading=)

# 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 core with hyperthreading up to 2
  + 4 performance core

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

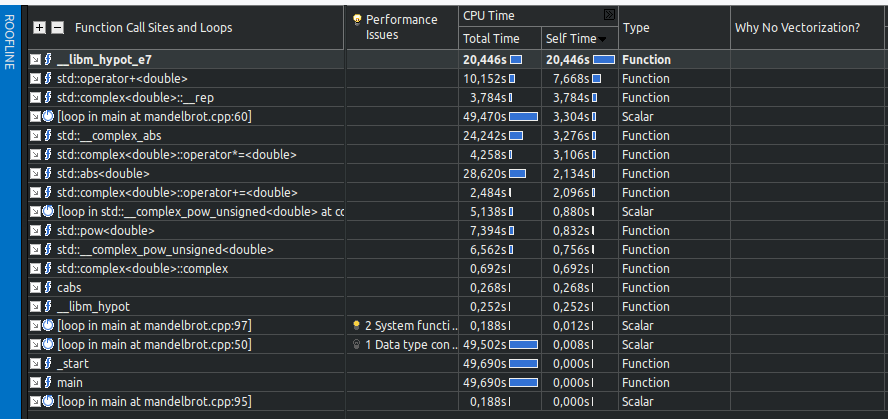
- **data size**: RESOLUTION=1000, ITERATIONS=700

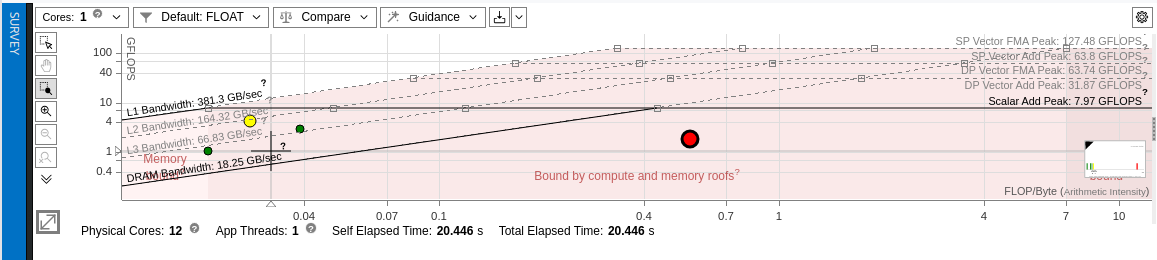
- **time taken**: 49.70 seconds

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.

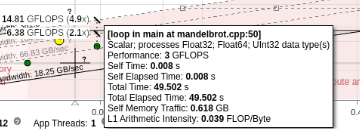
| **Name** | **Time taken (seconds)** |
| --- | --- |
| Overall program | 49.70 |
| \_libm\_hypot | 20.45 |
| operator+<double> | 7.67 |
| complex<double> | 3.79 |
| loop in main - line 60 | 3.78 |





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.
  + That may be caused by the fact that z is probably not a number but a pointer so the compiler can’t put it in a vector; since z can’t be transformed the outer loop fails to be vectorized
  + A possible solution, difficult to implement, could be to transform the coordinates from cartesian to polar, perform the exponential as a rotation and then transform from polar to cartesian again to allow an easier sum computation. We verified that this kind of transformation is vectorizable by the compiler, still we have a break that breaks the possibility of vectorization, since some z indexes should stop the for loop and should prevent the other indexes from being computed.  
    Indeed by removing the break we observed that the LOOP WAS VECTORIZED in the report. Still that cannot be safely removed, so not vectorization obtained.
* 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

# 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 -g -fopenmp mandelbrot.cpp

- **data size**: RESOLUTION=1000, ITERATIONS=7000

- **time taken**: 36.122 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=7000

- **time taken**: 21.7635 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:

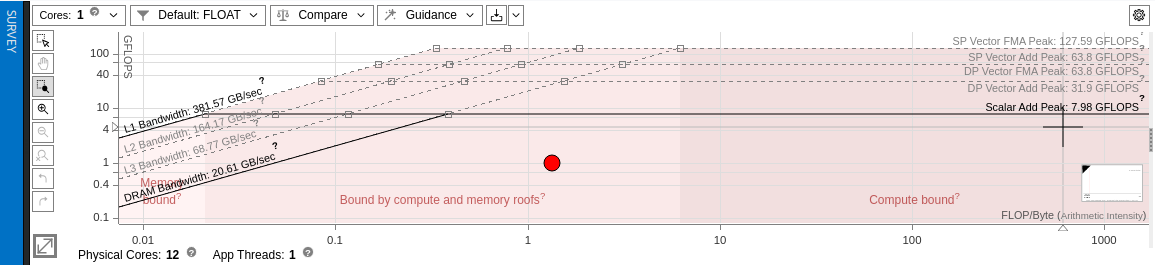
* **O3**: to allow the compiler to optimize the code, so that it can reorder the operations in the most efficient way
* **xHost**: to achieve the best assembly instructions specifically on the current machine architecture
* **ipo**: to enable interprocedural optimizations; we observed that with this 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.

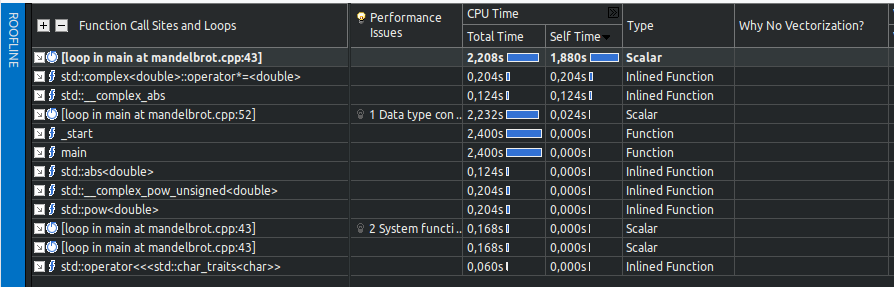
We also check on the roofline the result obtained so far with the combination of flags, also decreasing the data size for a matter of time spent by Intel Advisor:

- **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.

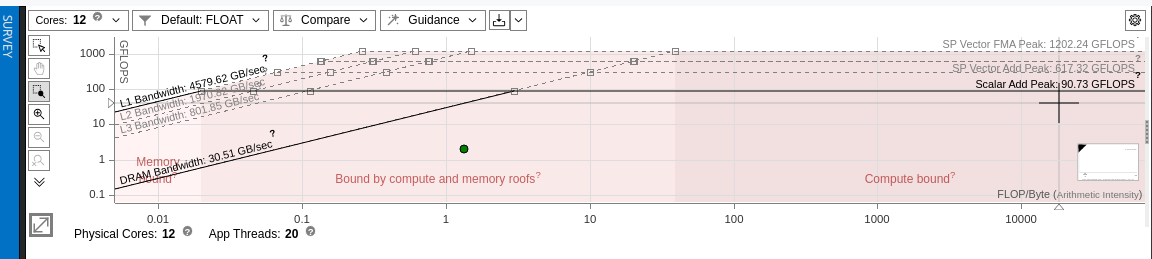
Now we increase the data size to test the limits by trying different combinations of the usage of the *pragma* instructions.

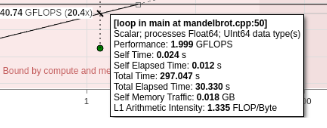
- **compiling line**: icx -fopenmp -O3 -xHost -ipo -ffast-math mandelbrot.cpp

- **data size**: RESOLUTION=1000, ITERATIONS=80000

- **number of threads**: 20 (decided by scheduler)

- **time taken**: 30.3726 seconds





We increased our arithmetic intensity. Altri commenti? ROOFLINE rotto 295 secs

# Speedup and efficiency - TODO

As possible number of processors, we chose:

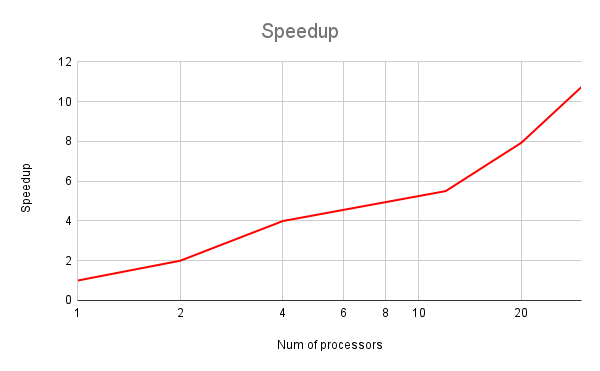
* 1: sequential run
* 2: enable parallelization
* 4: equal to the number of performance cores
* 12: equal to the number of all cores without considering hyperthreading
* 20: the maximum value of processors considering hyperthreading up to 2
* 30: to observe any overhead

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

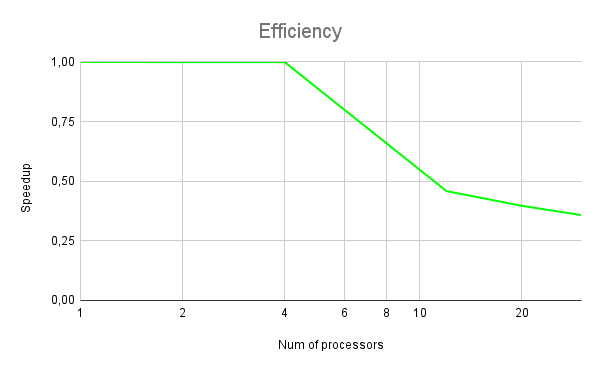
| **Data size** | **90000** |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| N processors | Time spent | Speedup | Efficiency |
| 1 | 33,17 | 1 | 1,00 |
| 2 | 16,6 | 2 | 1,00 |
| 4 | 8,3 | 4 | 1,00 |
| 12 | 6,03 | 6 | 0,46 |
| 20 | 4,18 | 8 | 0,40 |
| 30 | 3,09 | 11 | 0,36 |

# 

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 three different sections: after 2 processors (so enabling the parallelization), between 4 and 12, and then after 12 obtaining a huge speedup when exploiting the maximum number of processors.



On the other hand, the efficiency remains more or less stable up to 4 processors, then starts decreasing, observing that the percentage of decrement is bigger between the measures of 4 and 12 processors rather than the 12 and 20 measures. As expected continues to decrease also after that number.

# Conclusions

TODO

We observed that, in the best sequential case, our code performs the run in about 30 seconds, on an amount of data that has to be triplicated from the original unoptimized sequential run (that performed in about 60 seconds for N=30000). So the compilation optimization flags increased by a lot the performance of our code given also by the quadratic complexity, so the result obtained with the new amount of data indeed shows a huge improvement.

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 by using 4 we halved again.

The optimal number of threads to be used, without losing too much in terms of efficiency is around 10: so 12 in our choices. 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, also because the efficiency dramatically decreases.

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.