mandelbrot-CUCL

Eros Viola, 4219790@studenti.unige.it
Davide Riva, 4632421@studenti.unige.it

Introduction

The goal of this project is to build an image representation of a Mandelbrot set with as much resolution as possible.

A Mandelbrot set is a set of complex numbers for which the sequence $z_0 = 0$, $z_{n+1} = z_n^{(DEGREE)} + c$ does not diverge. It is proven that if $|z_n| > 2$ at some point n, that succession will never converge. If a particular sequence diverges, its corresponding pixel value will be the minimum number of iterations required to know that. Otherwise, 0.

Different values of *c* will lead to different sequences. To represent it graphically, we have chosen an interval *[MIN_X, MAX_X]* for the real part of *c* and an interval *[MIN_Y, MAX_Y]* for the imaginary part. The final image will have a width of $(MAX\_X - MIN\_X) \cdot RESOLUTION$ pixels and a height of $(MAX\_Y - MIN\_Y) \cdot RESOLUTION$ pixels.

The hardware

The cluster

It has 11 nodes, each node equipped with an Intel Xeon Phi 7210. Nodes are connected through a network (InfiniBand QDR, 40 Gbps).

Since Intel Parallel Studio 2017 is installed, the application will be compiled with it.

The GPU

4 NVIDIA Tesla P4 is used on a Google Cloud Platform machine (n1-standard-4, 4 vCPU).

The software

First, we present a sequential approach. Then, a parallel one with OpenMP and a distributed one with OpenMP+MPI. Finally, we show a version implemented with CUDA.

For simplicity, during performance evaluations, we always compute the set with *DEGREE = 2*. Each version has some parameters that will be tuned in a greedy approach.

The application is composed of two parts:

- **the core part**:
    - it is written in C++11;

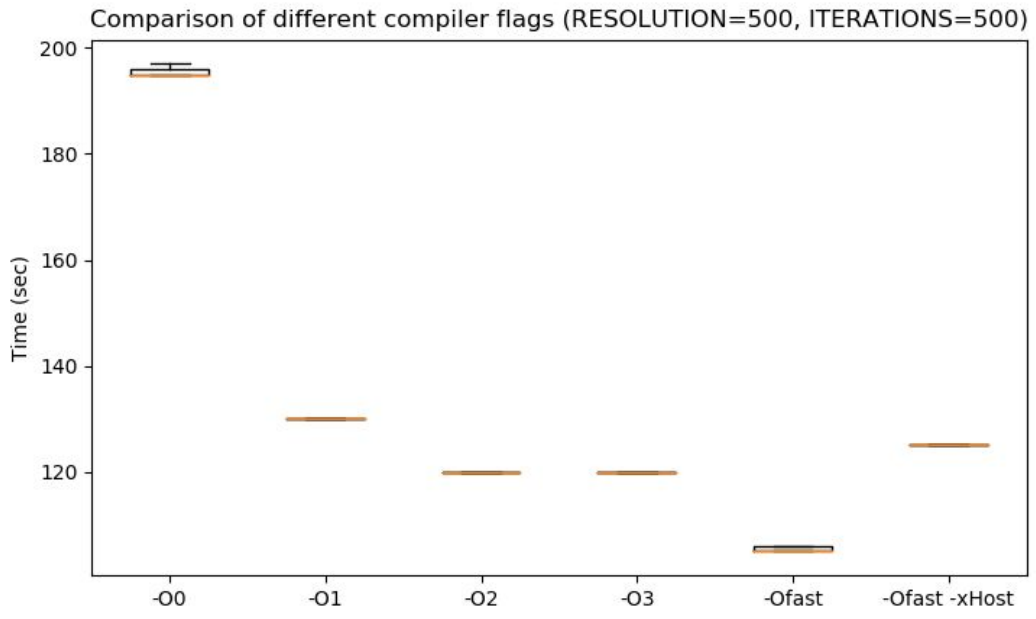
    - given a polynomial degree (*DEGREE*) and the maximum number of sequences iterations (ITERATIONS), it computes that particular Mandelbrot set;
    - it must be run in an HPC environment;
    - it saves the final result in a CSV (it's not counted during the evaluation of the performances);
- **the "visualization" part**:
    - it is a script written in Python;
    - given a final result from the previous part, it shows the graphical representation of it;
    - it can be run anywhere without worrying about the performances;

Both parts need the filename of the CSV as a parameter: the first one to create it, the second to read it.

Sequential version

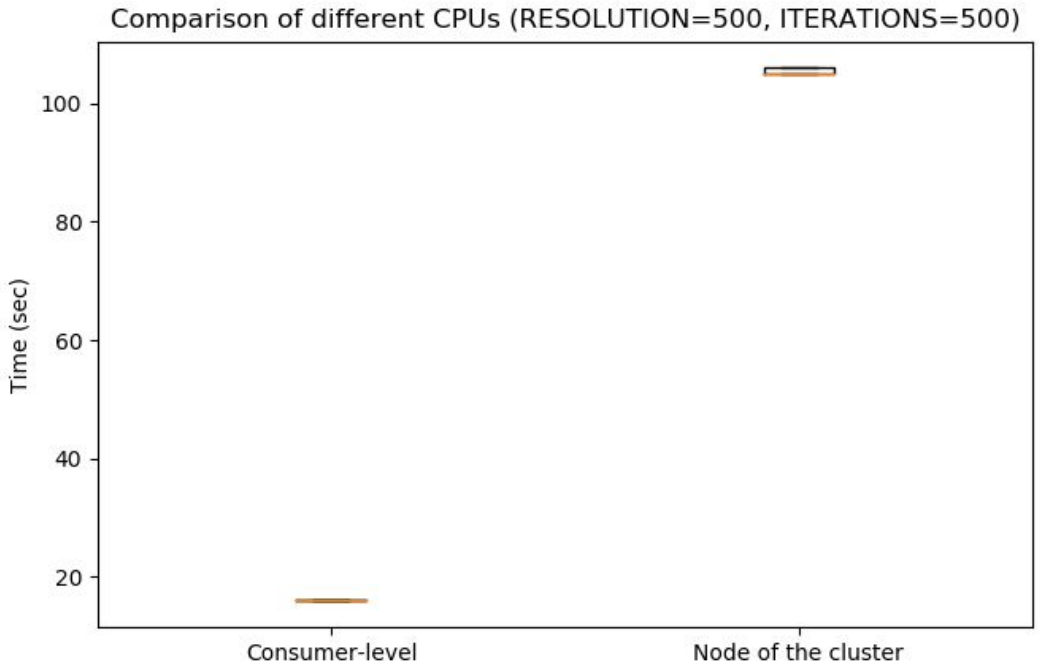
The sequential version can be found at *code/sequential.cpp*. Note that we prefer to use preprocessing macro definitions over constants because they are more performant: some computations are done at compile-time and also you can use faster assembly operations (e.g. *mov ax, bx* is slower than *mov ax, 3*).

Here we compare different ICC compiler flags in order to optimize the code in the best way:



As you can see from the plot, the best compiler flag in our scenario is *-Ofast*. Combining it with *-xHost* gives us a worse result than using *-Ofast* alone. That behaviour could be explained by the fact that no computationally expensive parts of our code could be vectorized.

An interesting thing to notice is that this version runs more quickly in a consumer-level computer with an Intel Core i3-7100U CPU (the source is compiled with g++ with no particular optimization flags) rather than in a single node of the cluster:



This is because the CPUs of the cluster are the best option (as you will see soon) when you have a highly parallelizable code, but they perform badly in a sequential and not vectorized scenario.

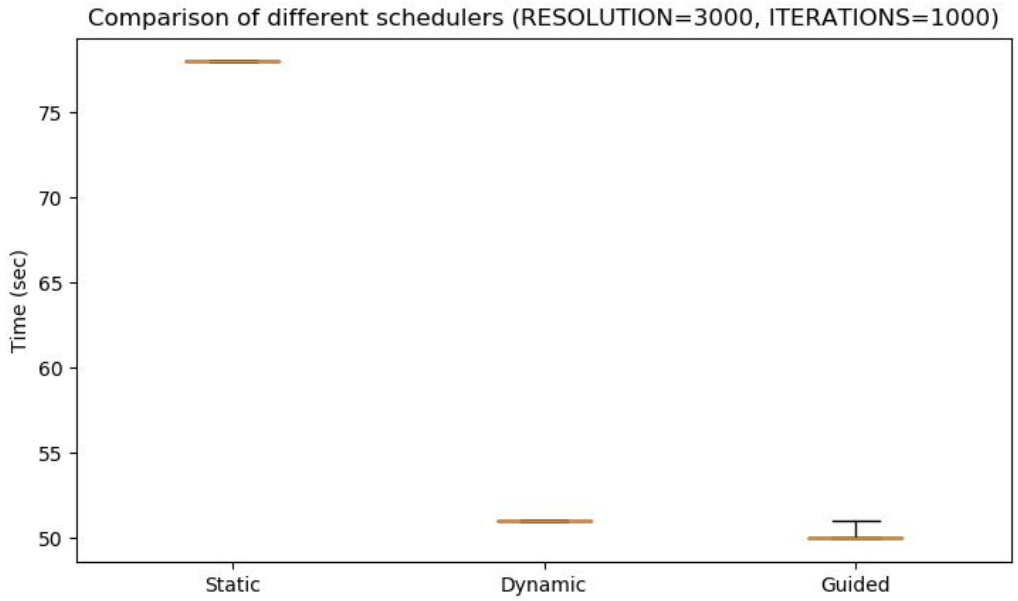
Note

Wait a minute, why are you not writing vectorizable code?
Vectorizing the code requires doing the maximum number of iterations for each sequence since branching in that portion of the code is not allowed. We know empirically that most of the pixel will converge rapidly, so we can reduce enormously the number of computations by not vectorizing the code. Also, with a vectorized version it is not possible to know what is the minimum n for which $|z_n| = 2$.

Version with OpenMP

All the OpenMP implementations of the algorithm can be found at *code/openmp/**.

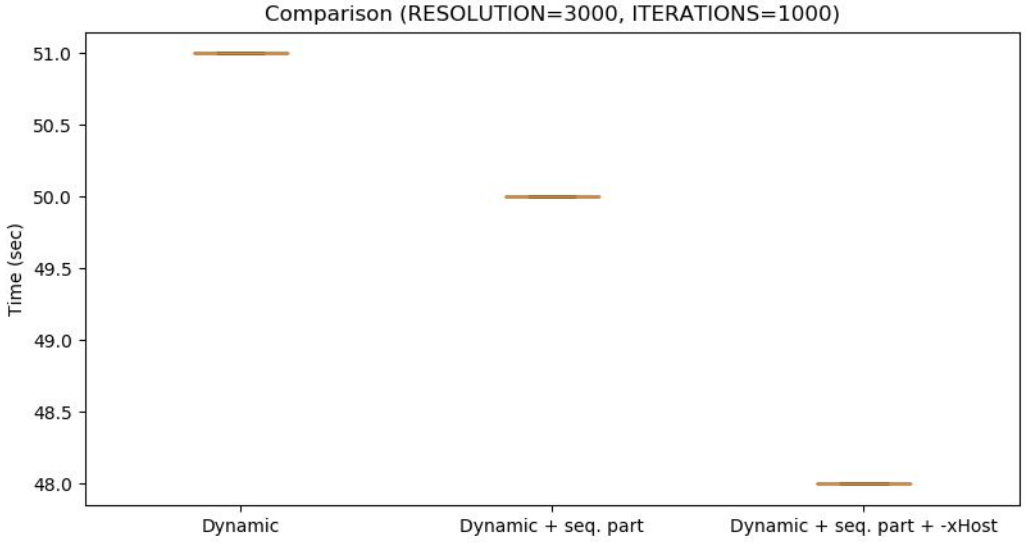
Three scheduling approaches will be considered: static, dynamic and guided.



As you can see from the previous plot, the dynamic and guided versions are the fastest one.

Since dynamic has also a theoretical reason to be the best one, we choose it among the two. The explanation is that each pixel has a different workload: different areas require different time to be computed. Note that we don't know a priori which areas require more computations and which ones require less.

Parallelizing this loop leads us to a bottleneck: in fact, a lot of threads try to write a value in the memory concurrently. Because of it, we move the initialization of the image in a sequential loop. Since that part could be vectorized (as intel-advisor told us), we re-tested the performance with and without *-xHost* as a parameter:



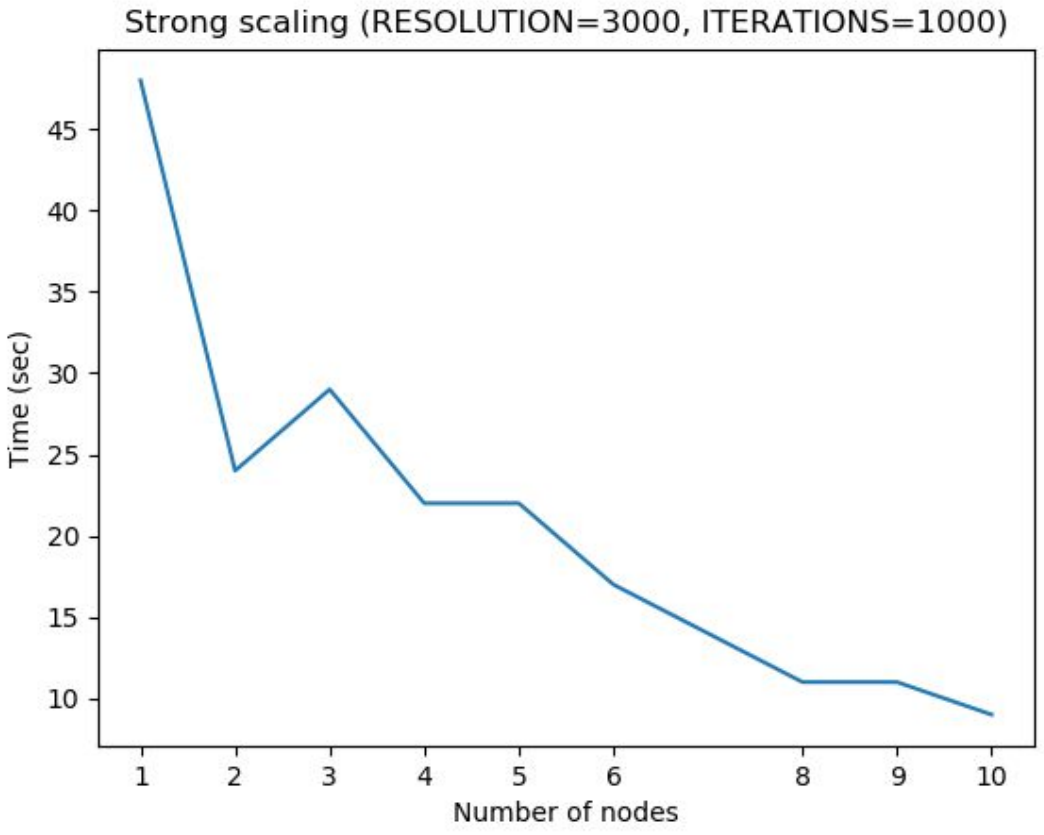
As expected, moving the initialization out of the parallel loop combined with the -xHost flag outperforms all the other variations.

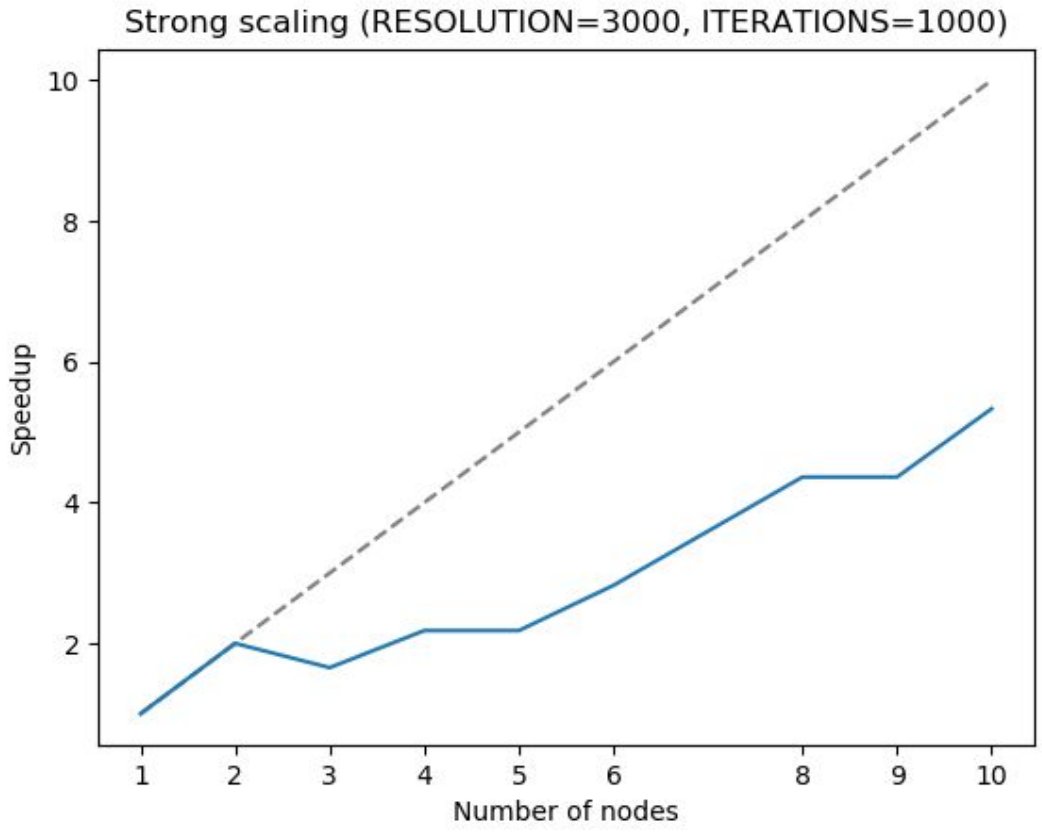
The last thing to do before moving to MPI is comparing the best parallel version with the sequential one. While fixing the same number of RESOLUTION (500) and ITERATIONS (500), we obtain that the best parallel version will be computed in less than a second. This means we have a boost in performance of more than *100%* (less than a second VS roughly 105-106 seconds).

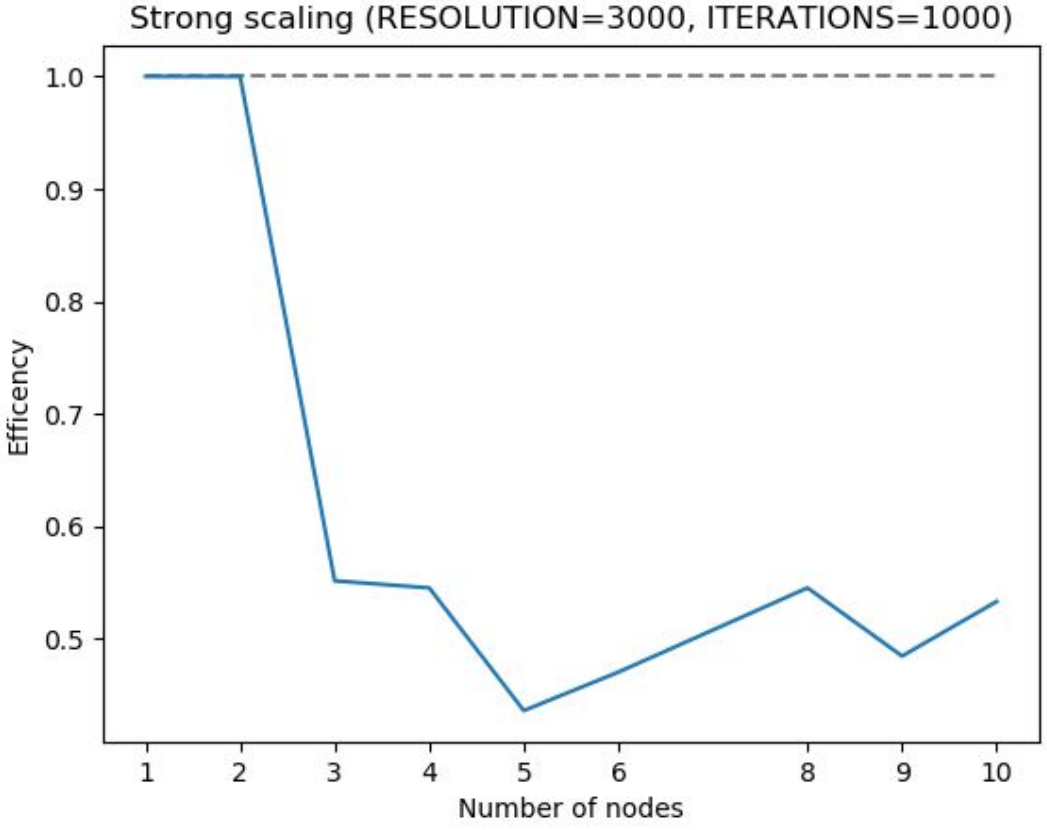
Version with MPI and OpenMP

An MPI implementation of the algorithm is proposed. Since MPI has a function (MPI_Wtime) to measure elapsed time, we use it instead of *chrono*. In this way, it is possible to use previous versions of C++ instead of C++11.

The code (*code/mpi/vanilla.cpp*) splits the image into contingent slices, each of them computed on a different node of the cluster. At the end it gathers all partial data into a node, called MASTER in the source code. This node has the duty to write the final matrix into a file.




Strong scaling (RESOLUTION=3000, ITERATIONS=1000)
Speedup
10
8
6
4
2
1
2
3
4
5
6
8
9
10
Number of nodes


Strong scaling (RESOLUTION=3000, ITERATIONS=1000)
Efficency
1.0
0.9
0.8
0.7
0.6
0.5
1
2
3
4
5
6
8
9
10
Number of nodes

From this plot, it is possible to infer that even if the performances are not monotonically decreasing since computations are not homogeneous among all nodes, keeping a high number of nodes seems a good idea.

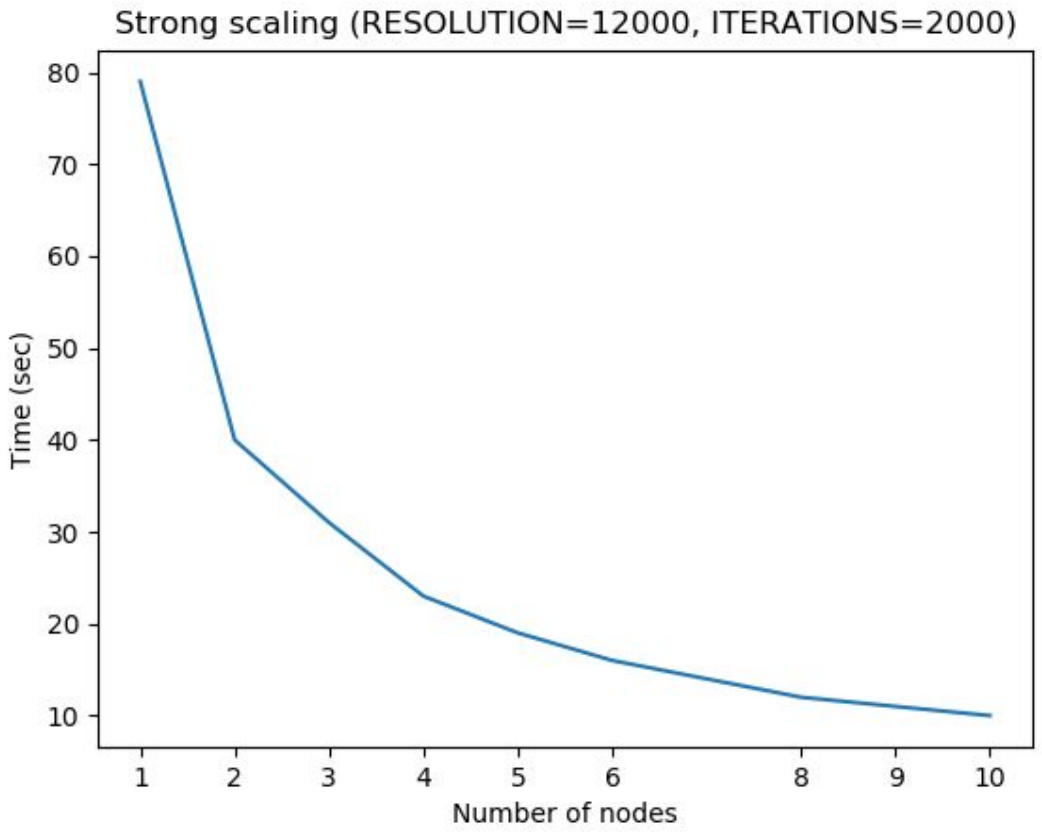
Comparing the best OpenMP-only version with this one, gives us that the new version is more than 5 time faster if *RESOLUTION=3000* and *ITERATIONS=1000*.

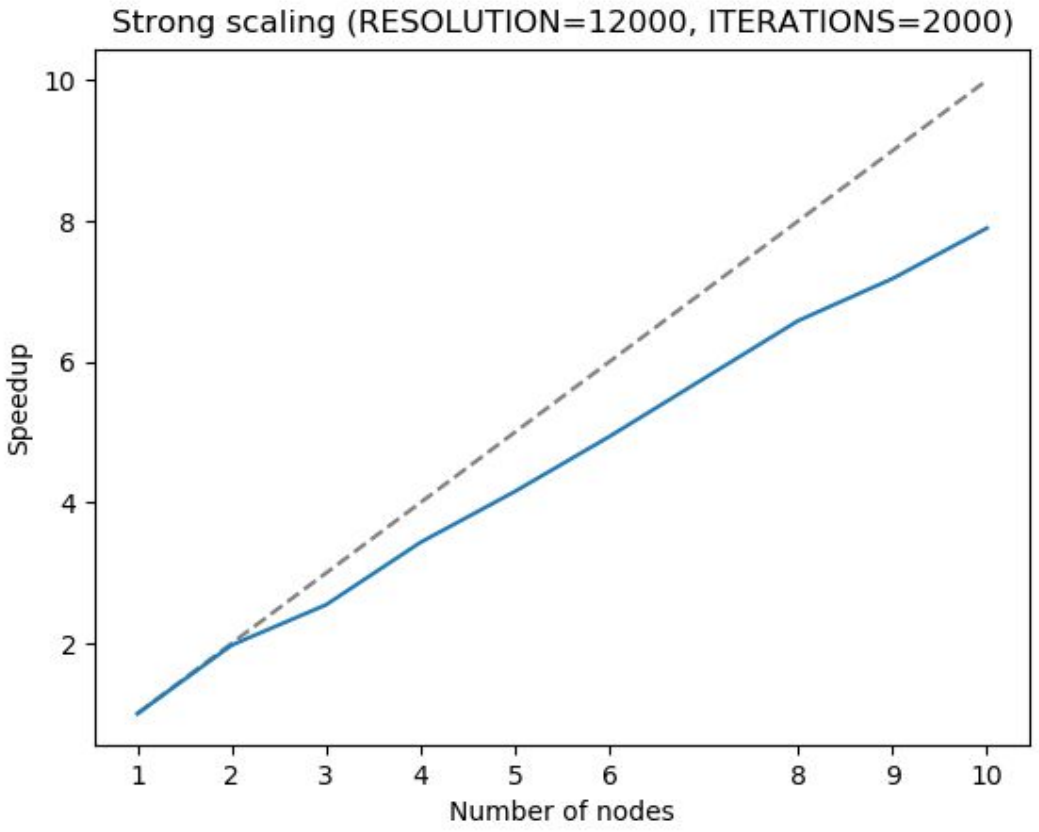
Version with MPI and OpenMP (degree fixed)

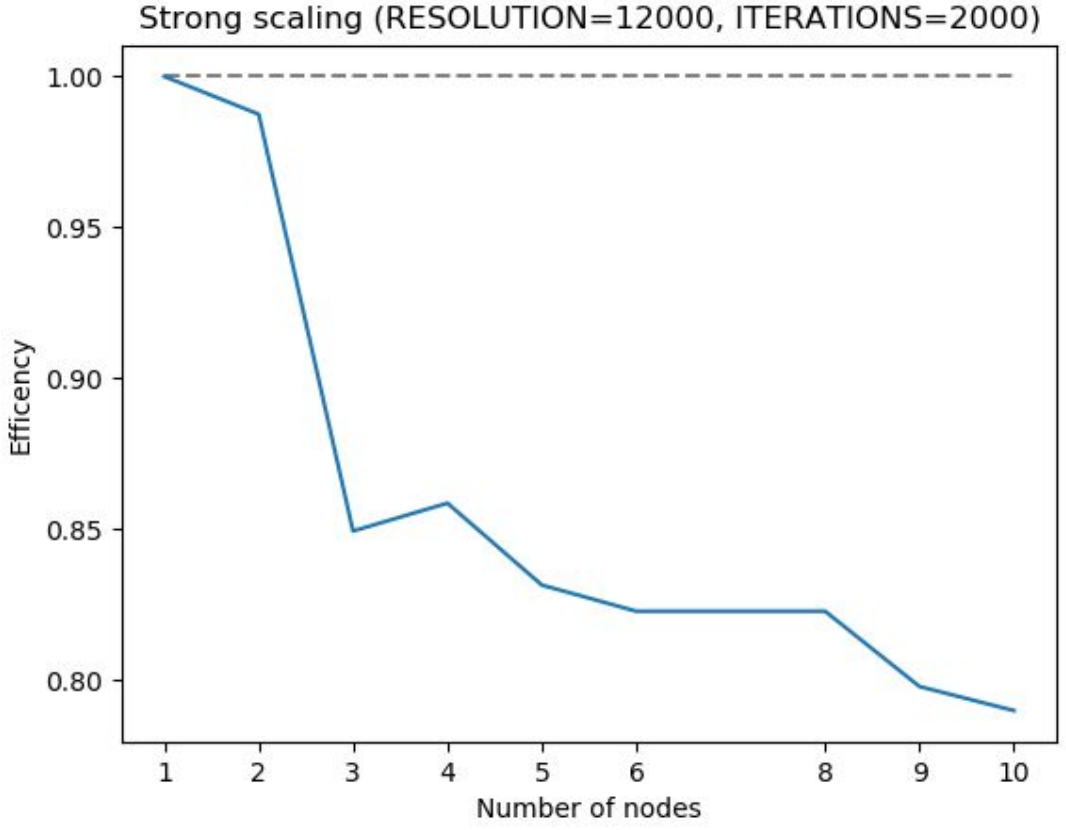
Since our goal is to squeeze the performances as much as possible, we decide to write a version in which the degree is fixed to 2 (*code/mpi/degree2.cpp*).

Thanks to these two formulas, we can simplify the computations related to complex numbers:

- $(a + ib)^2 = (a + ib)(a + ib) = a^2 - b^2 + i(2ab)$;
- $|a + ib|^2 = a^2 + b^2$

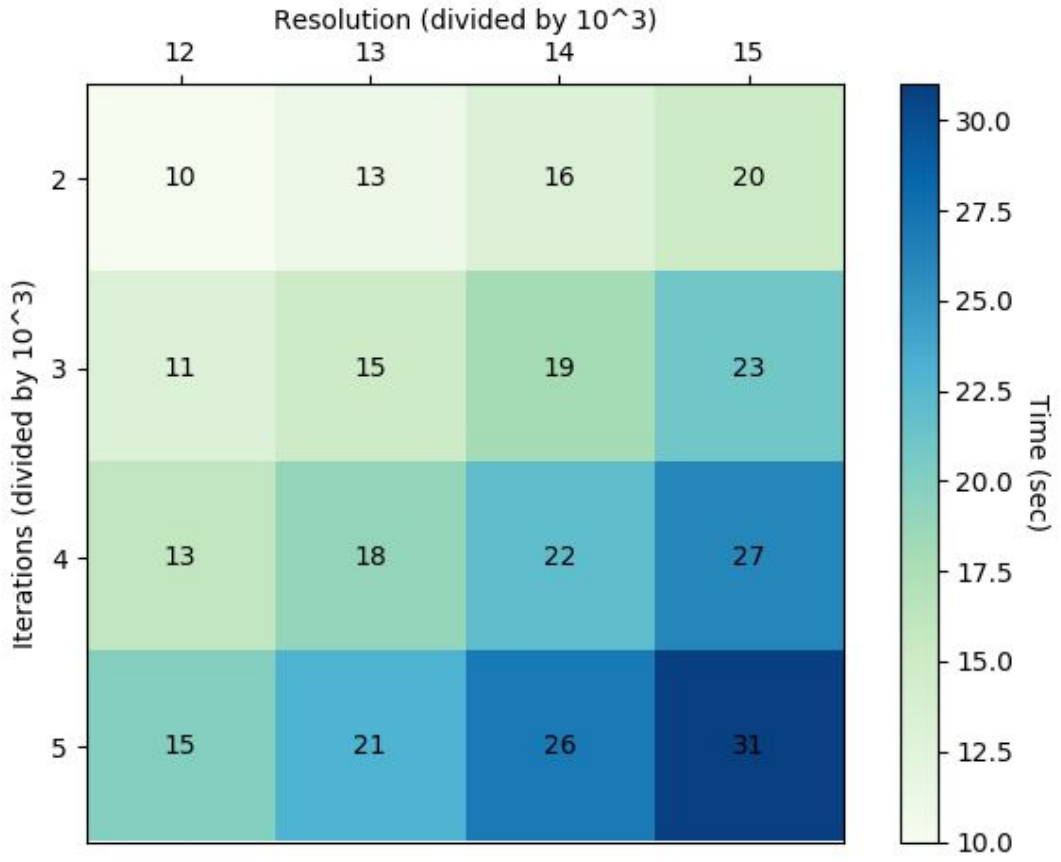



Strong scaling (RESOLUTION=12000, ITERATIONS=2000)
Speedup
10
8
6
4
2
1
2
3
4
5
6
8
9
10
Number of nodes


Strong scaling (RESOLUTION=12000, ITERATIONS=2000)
Efficency
1.00
0.95
0.90
0.85
0.80
1
2
3
4
5
6
8
9
10
Number of nodes

Since the workload can't be predicted from the number of pixels nor from the number of iterations, it's not possible to analyze the weak scaling factor.

Another way to see how it behaves under higher workloads is to plot a heatmap that considers different parameters values:



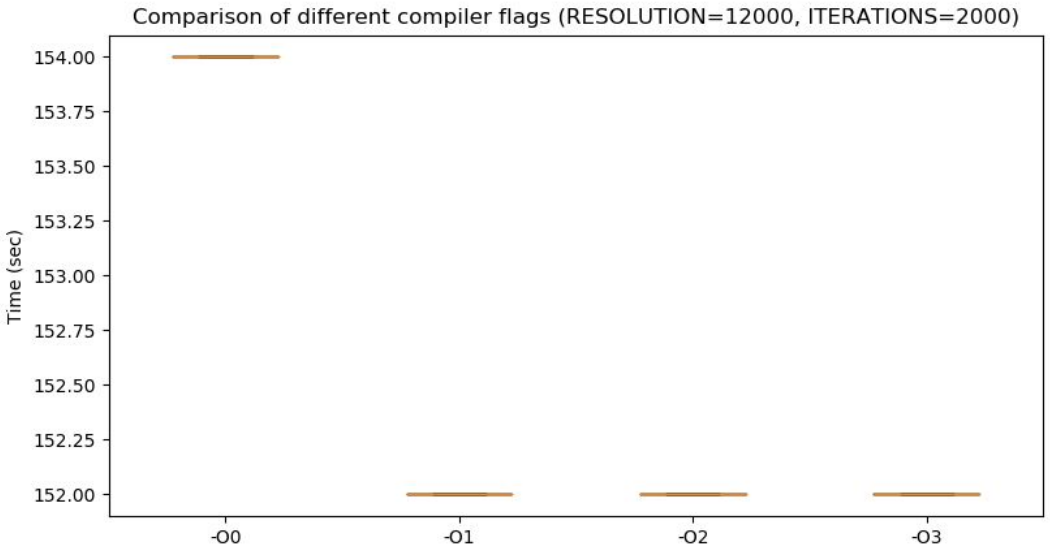
From the plot, it's clear that the time roughly grows linearly with the number of pixels and exponentially with the number of iterations.

The last thing to do before moving to CUDA is comparing the best vanilla version with the best new one. While fixing the same number of RESOLUTION (3000) and ITERATIONS (1000), we obtain that the new version will be computed in less than a second. This means we have a boost in performance of more than *10%* (less than a second VS 9 seconds).

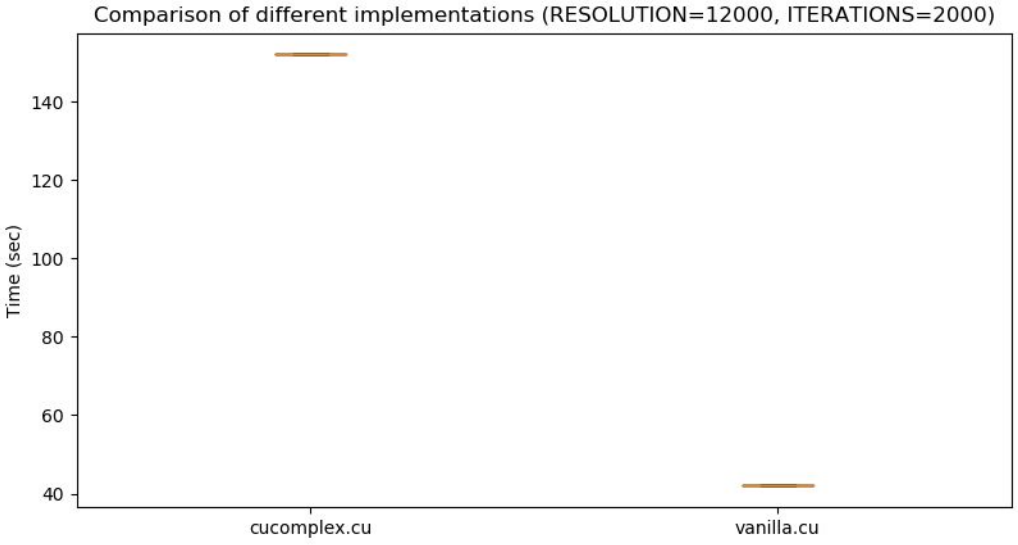
Version with CUDA

An implementation that targets Nvidia GPUs has been developed. The parallelization part is done instead of the sequential loop, as in the OpenMP version. Each pixel is computed by a different thread until that sequence converges or the maximum number of iterations is reached.

Since it is the official way, we use *cuComplex* to deal with complex numbers and operations between them (*code/cuda/cucomplex.cu*). Here there is an attempt to find the best optimization flags that target the host code:



Then, a comparison is made between it and our custom implementation (*code/cuda/vanilla.cu*) that uses doubles (as in the final version with MPI):



Finally, we analyze how the performances vary with changing the number of threads for each block:

RESOLUTION=12000, ITERATIONS=2000	
THREADS	Time (sec)
4x4	81
5x5	53
6x6	72
7x7	54
8x8	**42**
9x9	49

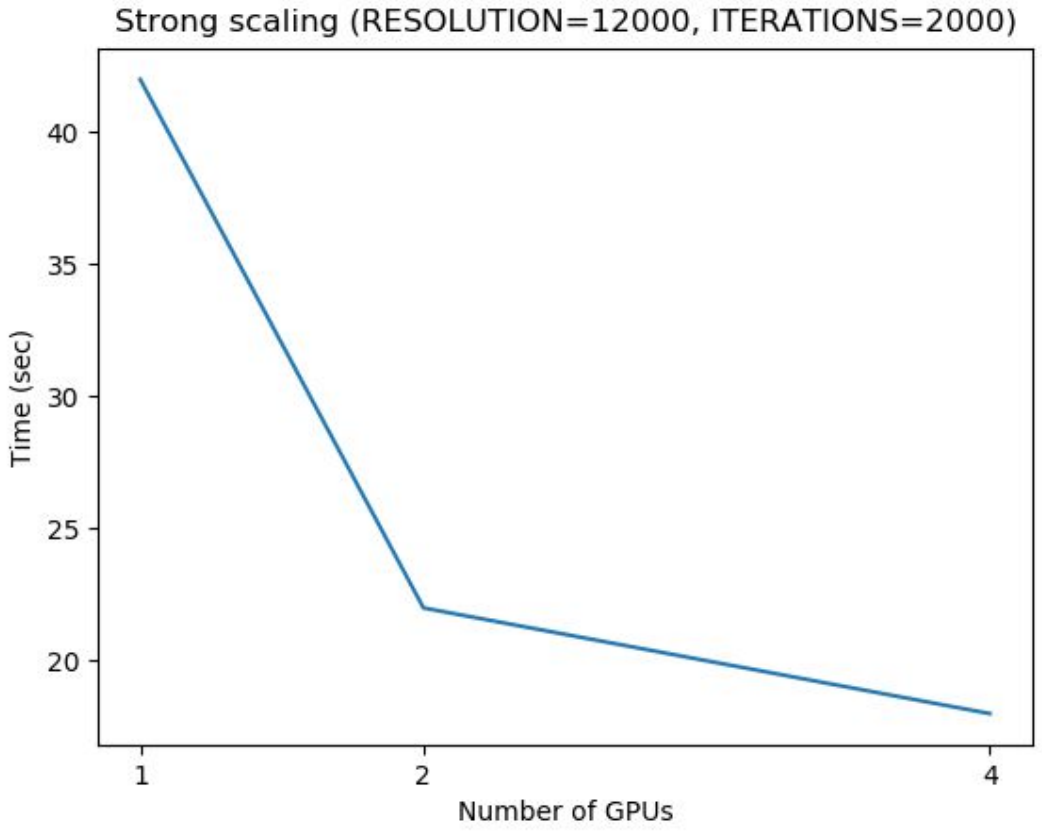
16x16	**42**
24x24	**42**
32x32	**42**

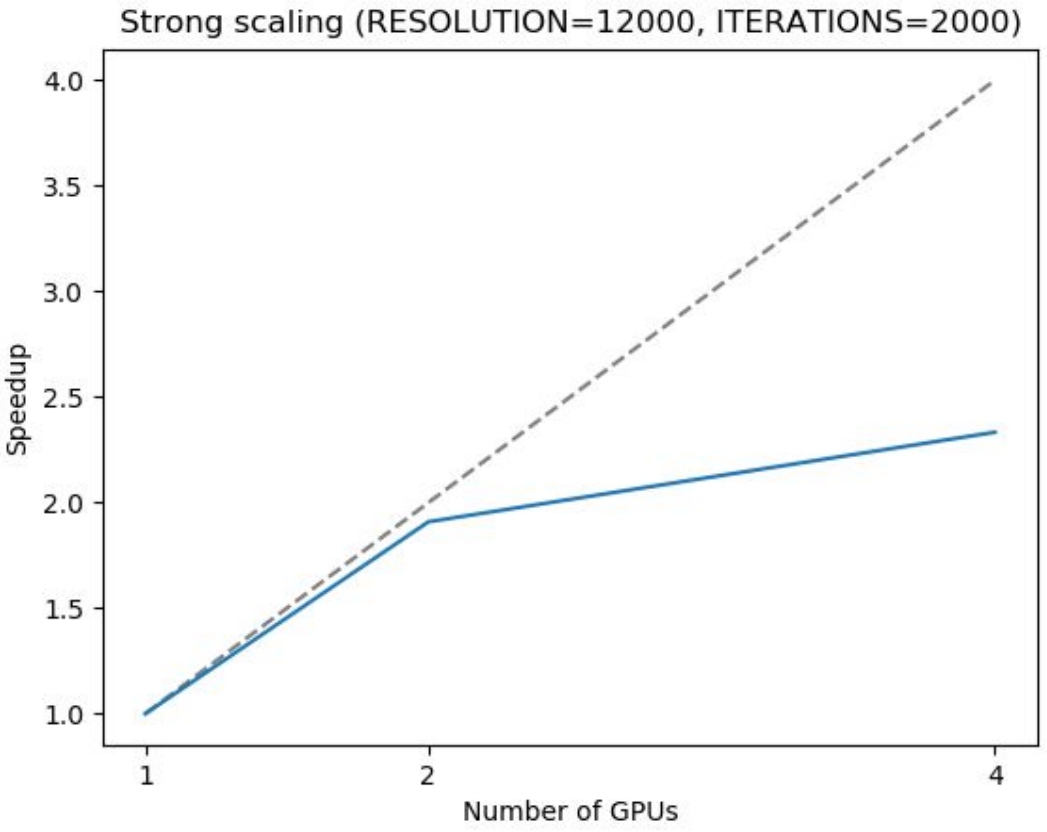
Since there are no shared variables inside each block, every number of threads per block which is a multiple of a warp size is a good choice. We pick the smallest one to avoid as much as possible the creation of useless threads which exceed the boundaries of the interval that we had chosen for rendering the set.

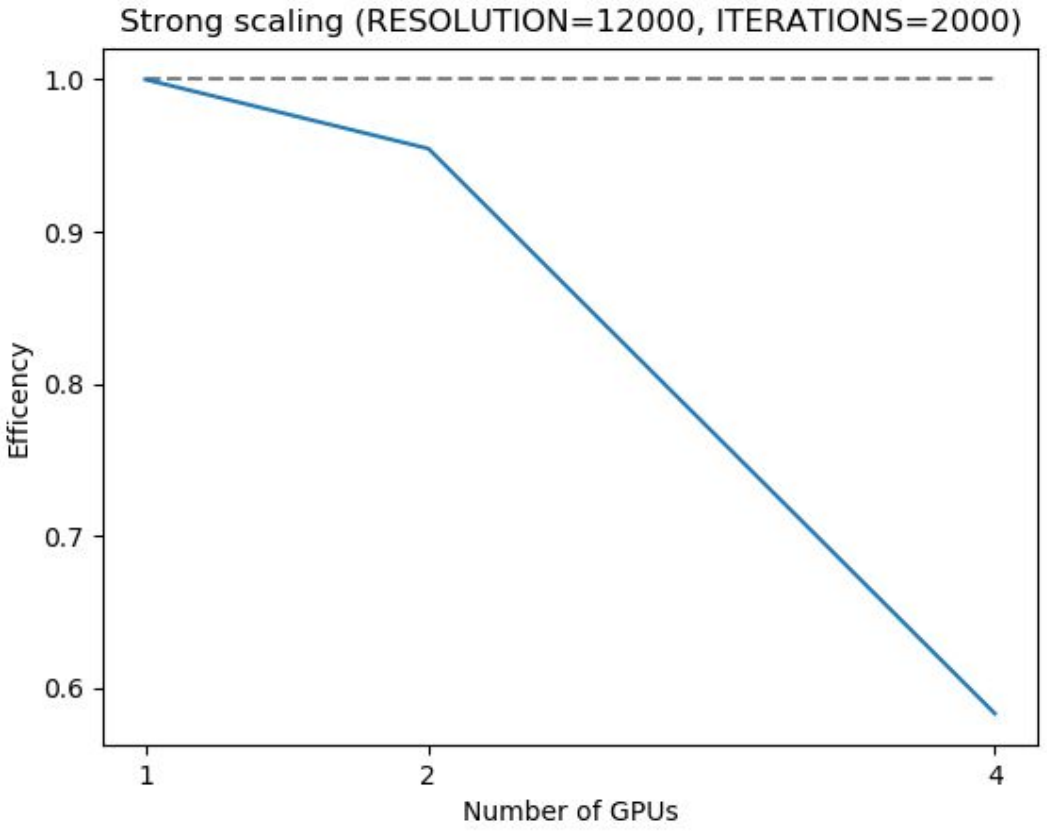
Compared to the OpenMP version, we obtain that the CUDA version is roughly 2x faster.

Version with CUDA (multiple GPUs)

We then tried to improve the speed of the computation by increasing the number of GPUs. The basic idea is that each GPU is assigned to a CPU thread. All GPUs do the computations asynchronously on a slice of the image.
We can compare it with the cluster of CPUs:




Strong scaling (RESOLUTION=12000, ITERATIONS=2000)
Speedup
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
Number of GPUs


Strong scaling (RESOLUTION=12000, ITERATIONS=2000)
Efficency
1.0
0.9
0.8
0.7
0.6
1
2
4
Number of GPUs

Even if the version with multiple GPUs outperforms the cluster version with the same amount of nodes / GPUs, the cluster still has the best performances since there are more nodes (12 nodes VS 4 GPUs). We stick to 4 GPUs since at this moment GCP doesn't allow you to increase the number of GPUs for each machine to more than 4 for our GPU model. Even if we will pick more pricey GPUs, the maximum number is still 8.

Additional notes

All plots are generated through a Python script (*code/plots.py*) which is in the project's repository.

All the math behind this project was taken from Wikipedia (https://en.wikipedia.org/wiki/Mandelbrot_set).