PROJECT PROPOSAL PHPC-2017

A High Performance Implementation of the Escape Time Algorithm to generate the Mandelbrot Set

Principal investigator (PI)	Concetto Emanuele Bugliarello
Institution	EPFL
Address	/
Involved researchers	Only PI
Date of submission	June 4, 2017
Expected end of project	June 4, 2017
Target machine	Deneb
Proposed acronym	MNDLBRT

Abstract

The Mandelbrot set is at the heart of fractal-like structures and many people have generated breathtaking drawings by sampling the complex numbers in it. In this project, we present a parallel implementation of the "escape time" algorithm, one of the algorithms used to produce such drawings. Our application is coded in C and makes use of the MPI standard and the CUDA platform. The source code is available in the c4science repository at: `https://c4science.ch/diffusion/3860/mndlbrt.git`

1 Scientific Background

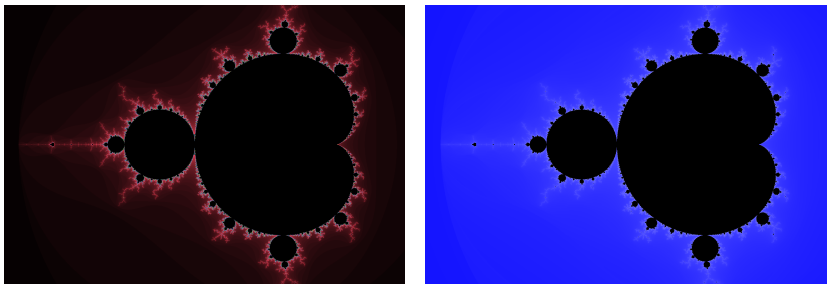
The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$ [1]. That is, if we denote the Mandelbrot set by M, then by repeatedly applying the quadratic map

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n + c, \end{cases}$$

for any complex number c, we have that $c \in M \iff \limsup_{n\to\infty} |z_{n+1}| \leq 2$.

To generate Mandelbrot set images, we can sample the complex numbers and iterate this function for each point c that has been sampled. If the function goes to infinity, then the point belongs to the set. Each sampled point can then be represented on a 2D plane by treating its real and imaginary parts as image coordinates ($x + yi$) and coloring the corresponding pixel according to how quickly the sequence $z_n^2 + c$ diverges.
Two examples of such representation are shown in Figure 1. In particular, points belonging to the set are colored in black, while points not in the set are colored according to how many iterations are required for the absolute values of the corresponding sequences to become greater than a cutoff value (2).

There exist many algorithms to draw the Mandelbrot set. Since our goal is to produce a parallel implementation in CUDA [2] and MPI [3], we use one of the simplest algorithms to generate a pictorial representation of the Mandelbrot set: the "escape time" algorithm. In fact, this algorithm results in an embarrassingly parallel problem: each pixel does not depend on any other, allowing each thread in the GPU to run its computations independently and avoiding exchanging ghost cells when MPI is used. The pseudocode of this algorithm follows.



(a) Red-shaded drawing. **(b)** Smooth, blue-shaded drawing.

Figure 1: Mandelbrot set drawings generated with our palettes.

Algorithm 1 Escape Time algorithm

for each pixel (Px, Py) on the screen **do**
    $x_0 \leftarrow$ scaled x coordinate of pixel
    $y_0 \leftarrow$ scaled y coordinate of pixel
    $x \leftarrow 0.0$
    $y \leftarrow 0.0$
    $iteration \leftarrow 0$
    while ($x \times x + y \times y < 2 \times 2$ AND $iteration < max\_iteration$) **do**
        $x_{temp} \leftarrow x \times x - y \times y + x_0$
        $y \leftarrow 2 \times x \times y + y_0$
        $x \leftarrow x_{temp}$
        $iteration \leftarrow iteration + 1$
    $color \leftarrow palette[iteration]$
    $plot(Px, Py, color)$

Where $z = x + iy$, $c = x_0 + iy_0$, $x = Re(z^2 + c)$ and $y = Im(z^2 + c)$.
Note that we keep iterating up to a fixed number of steps. Hence, we decide that a sampled point is "probably" in the Mandelbrot set after $max\_iteration$ iterations.

2 Implementations

The application is coded in C [4] and three main classes of implementations are developed:

- Serial versions.
- GPU versions using the CUDA parallel computing platform.
- Hybrid versions using both CUDA and MPI (including MPI-IO).

The code has been fully debugged using the `gdb` debugger. The `Valgrind` tool has been used to make sure that all heap blocks are freed and hence no memory leaks are possible.

2.1 Serial implementations

We start by implementing a naive version of the described algorithm and two plotting functions: a simple one with a red hue (Figure 1a), and a smoother version in blue[1] (Figure 1b). While the latter is prettier, it needs the real and imaginary values reached by each pixel in the quadratic map, thus requiring, in a first implementation, the allocation of two additional matrices of doubles (one for the real and one for the imaginary parts) of the same size as the total number of pixels. Throughout this project, we produce smooth, blue drawings as in Figure 1b.

[1] It uses an approximation of the Normalized Iteration Count method presented in [5].

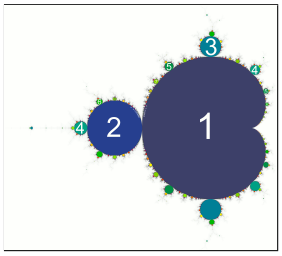


Figure 2: Periods of hyperbolic components [1].

Optimization level	Execution time [s]
`-O0`	422.91
`-O1`	230.99
`-O2`	208.96
`-O3`	208.49
`-O3 -ftree-vectorize`	205.41

Table 1: Execution times of `opti3` for a 25600×14400-pixel image and $max\_iteration = 10,000$ as a function of the optimization level.

Beside the standard optimizations (such as vectorization), we exploit the characteristics of the problem at hand in order to further improve the performance.

Firstly, as we can see from the pictures above, the resulting figure is symmetric with respect to the x axis. Then, we only compute the values for half of the specified height and fill the other half of the image during the plotting phase. We refer to this first optimization as `opti1`.

Secondly, it is possible to skip the calculations for the points lying within the cardioid or in the period-2 bulb (marked with 1 and 2, respectively, in Figure 2).
To do so, before passing a point to the escape time algorithm, we first check whether one of the following equations hold:

$$\begin{cases} q\left(q + \left(x - \frac{1}{4}\right)\right) < \frac{1}{4}y^2 \\ \\ (x+1)^2 + y+^2 < \frac{1}{16}, \end{cases}$$

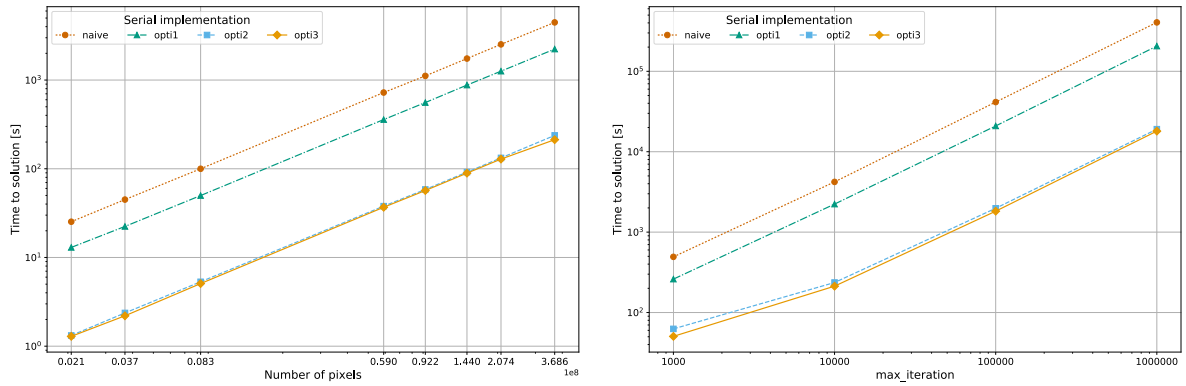
where $q = \left(x - \frac{1}{4}\right)^2 + y^2$, and x and y represent the real and the imaginary parts of the point. The first equation in the system determines if the point is within the cardioid, while the second one determines if it is within the period-2 bulb. The two shapes cover $\approx 34\%$ of the total area. 3rd- and higher-order bulbs do not have equivalent tests because they are not perfectly circular. We refer to the implementation using this optimization and the previous one as `opti2`.

Finally, we apply some finer-grained optimizations. In the above pseudocode, y is computed with two multiplications and one addition. Since multiplications are more expensive than additions, we would like to replace them by additions. Here, we can remove one multiplication in lieu of an addition: we firstly assign $x \times y$ to y, then we add y to itself (hence removing the multiplication by 2), and finally add y_0. Moreover, since we need x^2 and y^2 both in the `while` condition and in x_{temp}, we add two variables to store them so as to avoid recomputing two multiplications. By doing so, we only have three multiplications per loop, which is the minimum that can be achieved in this algorithm. We also employ fast array indexing while plotting the image. The implementation using all the described optimizations is denoted `opti3`.

The performances of the presented optimizations are compared in Figure 3. As we can see, each of these optimizations reduces the time to solution. In particular, we halve the time to solution when we evaluate the escape time only on half image, and a drastic reduction is observed when we avoid evaluating the escape time for the sampled numbers lying within the cardioid or in the period-2 bulb as each of them would reach $max\_iteration$ otherwise.

Each of serial implementations used in generating this figure were compiled using `gcc` and setting the compiler's optimization level to `-O3` with the loop vectorizer (`-ftree-vectorize`). In fact, this gives the best performance for each of them. As an example, Table 1 illustrates the execution times for `opti3` as it is compiled with different optimization levels.

We also compare the times to solution of `opti3` both when the resulting image is drawn and



(a) **(b)**

Figure 3: Time to solution for the different serial implementations with respect to (a) the number of pixels (max_iteration=10,000) and (b) max_iteration (368,640,000 pixels).

when the execution stops just after evaluating the color of each pixel: the two curves basically overlap suggesting that the writing time is negligible.

3 Prior results

We run our implementations on the Deneb cluster at EPFL, whose specifics are [6]:

- 376 compute nodes, each with (i) 2 Ivy Bridge processors running at 2.6 GHz, with 8 cores each, and (ii) 64 GB of DDR3 RAM.
- 144 compute nodes, each with (i) 2 Haswell processors running at 2.5 GHz, with 12 cores each, and (ii) 64 GB of DDR4 RAM.
- 16 GPU accelerated nodes, each with 4 K40 NVIDIA cards.
- Infiniband QDR 2:1 connectivity.
- GPFS filesystem.
- Total peak performance: 293 TFLOPs (211 in CPUs, 92 in GPUs).

3.1 Theoretical computational cost

In this section, we try to estimate the computational cost of the escape time algorithm. In particular, we consider the `opti3` implementation of the algorithm.
For the sake of simplicity, let's consider generating square figures, where width and height have N pixels. Due to the previous symmetry optimization, the algorithm is called $N \times \frac{N}{2}$ times.

An approximation of the theoretical cost can be obtained by considering a worst-case scenario where each sampled complex number that is neither in the cardioid nor in the period-2 bulb iterates *max_iteration* times.

In evaluating the computational cost, we assume the following costs per operation [7]:

- addition, subtraction, comparison (1);
- multiplication, division by 2 (4).

In our implementation of the escape time algorithm, we have:

- 5 sums, 4 multiplications and 2 divisions by 2 to initialize the variables of each pixel;
- 5 sums, 1 subtraction, 2 comparisons and 5 multiplications to check if the pixel is within the cardioid or in the period-2 bulb;
- 5 sums, 1 subtraction, 2 comparisons and 3 multiplications in each iteration.

Note that these are simple approximations and the real cost of each operation actually depends on the computer architecture. Furthermore, we are ignoring the cost of conditional statements which lead to expensive branch mispredictions.
That said, an estimation of the computational cost for an $N \times N$-pixel image is then given by:

$$\approx (c + 0.76 \times max\_iteration) \times \frac{N^2}{2}$$

Where 0.76 is the percentage of pixels that are neither in the cardioid nor in the period-2 bulb and $c \approx 57$. Hence, the algorithm's computational cost is $O(N^2)$ and so linear in the number of pixels (as we expected from its formulation).

Note that this cost does not take into account the expensive operation to determine the smooth color for each pixel and the additive time needed to store the matrix of iterations as an image, which are also $O(N^2)$. Memory requirements are in the order of $O(N^2)$ too.

Finally, if we fix the size of the image, then the computational time increases linearly with the value of *max_iteration*. All these relationships are confirmed by the curves in Figure 3.

3.2 Strong scaling

Strong scaling means using parallel computing to run a problem faster than on a single core.
Strong scaling is usually equated with Amdahl's Law, which specifies the maximum speedup that can be expected by parallelizing portions of a serial program:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} = \frac{T_1}{T_p}$$

Where α is the fraction of non-parallelizable code, p is the number of processors, and T_1 and T_p are the execution times on one and p processors, respectively.

Hence, we need to estimate the fraction of non-parallelizable code α. The core of the algorithm is intrinsically parallel because of the independence of the computations between the pixels. However, serial sequences are present as well to (i) allocate and deallocate the variables, (ii) set the corresponding number of iterations in the other half of the image and (iii) store the resulting image onto disk. The impact of the serial part will be more and more evident as the size of the image (number of pixels) increases.
We use `gprof` to obtain an estimation of the value of α. In particular, we do this when drawing both a 4K image ($8,294,400$ pixels) and a 25600×14400 image ($368,640,000$ pixels). In both cases we set *max_iteration* to $10,000$. The value of α is estimated to be 0 in the former case, while equal to 1.91 in the latter; hence suggesting a perfect speedup for relatively small images. Figure 4 shows the expected speedup in these two scenarios.

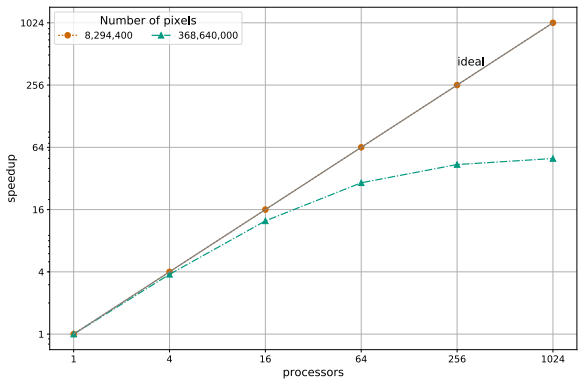


Figure 4: Expected speedup for small ($\alpha = 0\%$) and large ($\alpha = 1.91\%$) images from Amdahl's law.

3.3 Weak scaling

Weak scaling is what we refer to when we are talking about using parallel computing to run a larger problem in the same amount of time as a smaller one on a single core. In other words, it is a measure of how the time to solution changes as more processors are added to a system with a fixed problem size per processor.

We expect a significant speedup also in this case, but we are somehow limited by the size of the images we can create. So, we can instead increase the complexity of the problem by choosing larger values of $max\_iteration$. Knowing that the peak number of floating-point operations per second in one TESLA K40 GPU card is 1.43 $TFLOPs$ for double-precision and 4.29 $TFLOPs$ for single-precision [8], while having a peak of $\frac{211}{376\times2\times8+144\times2\times12} \approx 22.3$ $GFLOPs$ in a single core of CPU in the Deneb cluster, we can predict solving problems that are ≈ 66 (double-precision) or ≈ 197 (single-precision) times larger on the GPU in the same time as a reference problem on a single CPU core. Despite this wide gap, we should remember that these are theoretical predictions and several issues will actually limit the parallel performance, such as memory communication overheads and load balancing.

4 Going parallel

After having debugged, profiled and optimized our serial code, we can proceed with the parallelization phase. In fact, since each element in the array is independent of all the others, our algorithm leads to an embarrassingly parallel formulation.

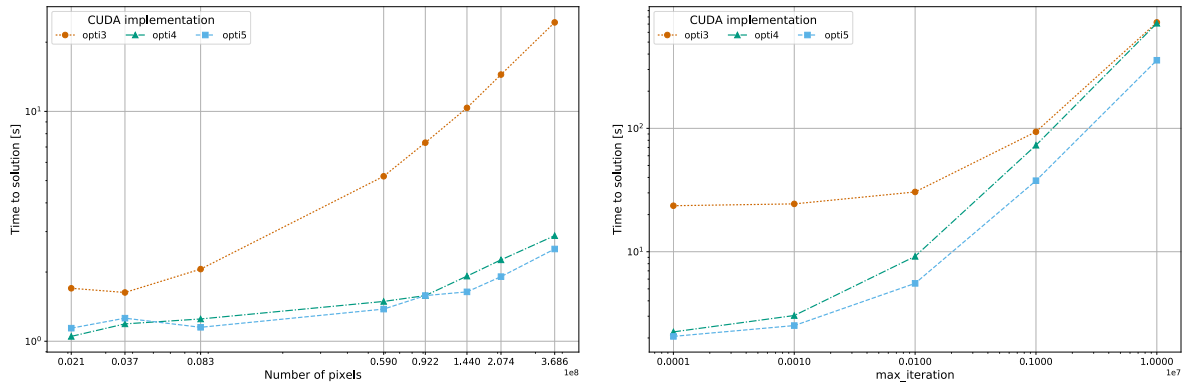
4.1 CUDA implementations

We start parallelizing our code with CUDA; once this version is optimized, we will add MPI.

Firstly, we just port `opti3` to CUDA. We do so by parallelizing the core of the algorithm: that is, evaluating the escape times for each of the points in the bottom half of the figure. This version is still far from optimal: most of the time spent by the serial code is in computing the color associated to a given iteration value and this part is still done serially. Moreover, we still use three matrices: one for the escape times, one for the real parts reached by the quadratic mappings and one for the corresponding imaginary parts. Sending three matrices on the PCIe bus surely does not lead to high performance computing, but this version serves as a first working CUDA implementation of the algorithm.

After debugging this version, we proceed implementing a new one: `opti4`. In this version, all the workload is carried out by the GPU. The only values exchanged by the between CPU and the GPU are the chars representing the RGB values for each pixel in the bottom half of the image. The code is split into serial part (`.c`) and parallel part (`.cu`), compiled with `gcc` and `nvcc`, respectively, and then linked together into one executable. To do so, we need a C wrapper for the kernel that allows the CPU to share parameters with the CUDA part. Moreover, all the constant values are copied to the Constant memory: a read-only, quick cache that can broadcast to all active threads.
When we profile this application with `nvprof`, we notice that it is compute-bound: that is, its arithmetic intensity is underneath the peak performance ceiling in the Roofline model.

Finally, we move from a double-precision to a single-precision representation with `opti5`. Strictly speaking, the resulting drawings are different from the previous ones but no human being would be able to tell the difference between two images produced using double-precision



(a) **(b)**

Figure 5: Time to solution for the CUDA implementations with respect to (a) the number of pixels (max_iteration=10,000) and (b) max_iteration (368,640,000 pixels) with block size 512×1.

and single-precision operands. We make this shift to single precision in order to maximize instruction throughput (as we described above and it is suggested in [9]).
The resulting optimization is still compute-bound.

The performances of the presented CUDA optimizations are compared in Figure 5.

We complete our study on CUDA by tuning the *block size* used by `opti5`. This is a crucial hyperparameter that leads to performance improvements and it is determined by both the problem at hand and the specifics of the used GPU. A properly set block size leads to increased occupancy; i.e., the ratio between the number of active warps per multiprocessor and the maximum number of active warps. Table 2 presents the achieved occupancy as a function of the block size for two image sizes when launched on Deneb. The winning size is `256×1`. We make use of `nvprof` to evaluate the occupancy. Even though larger image sizes and higher *max_iteration* would lead to a more accurate evaluation of the achieved occupancy, we had to set *max_iteration* to only 10,000 in order for `nvprof` to evaluate this metrics. Nevertheless, the highest achieved occupancy is close to 66%, which is enough to usually saturate the bandwidth [10]. Further, one should look at increasing occupancy only if the the kernel is bandwidth-bound, which is not our case.

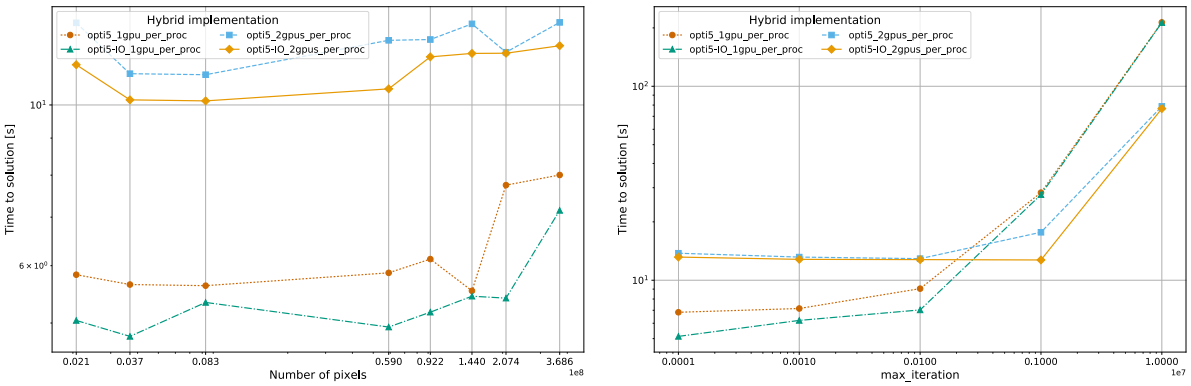
Block size	**Occupancy** [%]	
	2560x1440	25600x14400
32x1	18.05	18.81
64x1	25.41	32.65
128x1	31.82	51.52
192x1	34.85	56.01
256x1	**36.15**	**58.80**
384x1	35.62	57.30
512x1	28.76	52.37
1024x1	8.39	25.50
8x8	19.04	20.12
16x16	26.91	25.93
32x32	5.91	7.74

(a)

Block size	**Execution time** [s]	
	2560x1440	25600x14400
32x1	2.53	79.61
64x1	2.29	50.32
128x1	2.10	37.52
192x1	2.05	36.56
256x1	**2.04**	**35.91**
384x1	2.11	36.60
512x1	2.17	37.69
1024x1	5.65	90.60
8x8	4.09	237.72
16x16	2.45	97.12
32x32	7.48	270.24

(b)

Table 2: (a) Occupancy and (b) execution time for two image sizes as the block size varies.



(a) **(b)**

Figure 6: Time to solution for the hybrid implementations with respect to (a) the number of pixels (max_iteration=10,000) and (b) max_iteration (368,640,000 pixels) with block size 256×1.

4.2 Hybrid implementations

After having optimized on a single GPU, we introduce an additional layer of parallelization: we distribute the computations over different CPUs and GPUs with MPI. Specifically, we split the bottom half of the image to all the reserved GPUs by rows, and so each GPU receives the same number of pixels [2].
Each GPU node on Deneb has two processors and four GPUs. We then distribute `opti5` so as to make use of the resources available on a single node; we do so with two versions.

Firstly, we distribute the image to the two processors and each of them uses one GPU to evaluate the escape times and the corresponding colors. We provide two implementations: a first implementation using non-blocking, point-to-point communications where the processor with rank 0 receives the image parts in an out-of-order fashion. In particular, no `MPI_Barrier()` is used. The second implementation, instead, makes use of parallel I/O to increase the writing performance on the GPFS parallel file system.

Secondly, we distribute the image to the two processors and each of them uses two GPUs. That is, each processor is in charge of half of the bottom half image, which is then further split to the two GPUs. In this scenario, to achieve the highest performance, all the CUDA functions are asynchronous (with no barrier or thread synchronization used). Also this version comes with both the centralized, non-blocking implementation and the one using parallel I/O.

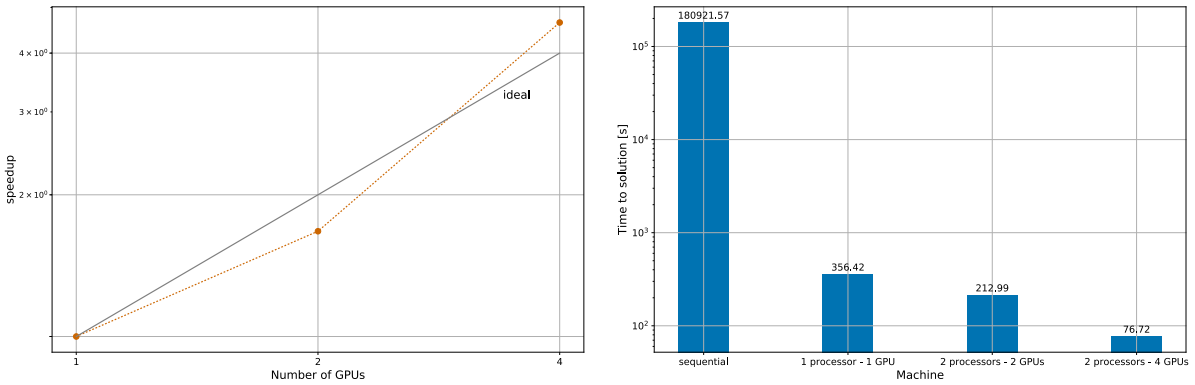
The performances of the hybrid implementations are shown in Figure 6, where we can see that the benefits of using multiple GPUs is manifested only for massive loads.

5 Final results

In this section, we present the results we obtained for scaling when CUDA and MPI are used. The results using multiple GPUs versions are obtained from the parallel I/O implementations.

Figure 7 presents the results for strong scaling: specifically, for a 25600×14000-pixel image and $max\_iteration = 10,000,000$.
Figure 7a shows how increasing the number of GPUs affects the speedup: here, we can see that the speedup obtained using four GPUs is higher than the ideal one. This can be explained by

[2] If dividing the number of rows by the number of GPUs does not return an integer, the remaining rows are assigned as uniformly as possible among all the GPUs.



(a) **(b)**

Figure 7: Strong scaling as (a) the speedup for an increasing number of GPUs (via MPI) and (b) the reduced time to solution of the same problem (25600×14000-pixel image, $max\_iteration = 10,000,000$) for the different implementations.

the fact that all the CUDA functions are called asynchronously in this version and both the multi-GPU versions use two processors.
Figure 7b illustrates how the time to solution varies from the serial version to the hybrid version.

Finally, to evaluate the effect of weak scaling, we fix the image size to $368,640,000$ pixels and determine the value of $max\_iteration$ that allows each version to execute in around 76 seconds. The results are reported in the figure below.

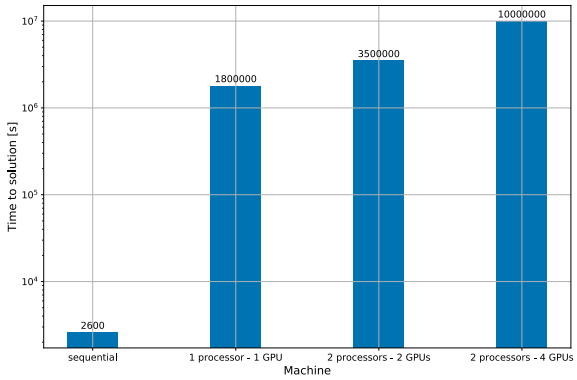


Figure 8: Weak scaling as a function of the problem size that can be solved in the same amount of time by the different implementations.

6 Resources budget

In order to fulfill the requirements of our project, we present hereafter the resource budget.

6.1 Computing Power

The parallel implementations are compute-bound; hence, powerful GPUs are required.

6.2 Raw storage

The main requirement in terms of raw storage is due to the output images. The largest image that we generate is 2.1 GB.

6.3 Grand Total

Total number of requested GPUs	4
Temporary disk space for a single run	3 GB
Permanent disk space for the entire project	50 GB
Communications	MPI
License	MIT
Code publicly available ?	Yes
Library requirements	CUDA & MPI
Architectures where code ran	Deneb

7 Scientific outcome

The aim of this project is to gain experience and expertise on CUDA and MPI while developing applications in a supercomputer. Throughout the development process, we have acquired familiarity with different tools for profiling and optimizing serial and parallel applications, as well as different technologies (such as non-blocking communications and parallel I/O in MPI). All the skills that we have mastered in this context will constitute a fundamental basis for real submissions in larger high performance computing centers.

References

[1] Wikipedia, *Mandelbrot set*, Online - accessed 09-May-2017, https://en.wikipedia.org/wiki/Mandelbrot_set

[2] NVIDIA, *CUDA: Parallel Programming and Computing Platform*, Online - accessed 09-May-2017, http://www.nvidia.com/object/cuda_home_new.html

[3] The MPI Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, 1994

[4] Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language Second Edition*, Prentice-Hall, Inc.,1988

[5] Stack Overflow, *Improvement to my Mandelbrot set code*, Online - accessed 09-May-2017, http://stackoverflow.com/questions/16124127/improvement-to-my-mandelbrot-set-code

[6] Scientific IT and Application Support, *Deneb and its extension, Eltanin*, Online - accessed 14-May-2017, http://scitas.epfl.ch/hardware/deneb-and-eltanin

[7] Vincent Hindriksen, *How expensive is an operation on a CPU?*, Online - accessed 14-May-2017, https://streamcomputing.eu/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/

[8] NVIDIA, *NVIDIA TESLA GPU ACCELERATORS*, Online - accessed 1-June-2017, http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf

[9] NVIDIA, *CUDA C Programming Guide*, Online - accessed 1-June-2017, http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4j1NpsHeT

[10] NVIDIA, *CUDA Warps and Occupancy*, Online - accessed 1-June-2017, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf