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

Parallel and high-performance computing

Emanuele Bugliarello
emanuele.bugliarello@epfl.ch
(SC-MA4)

June 29, 2017
Table of contents

- Scientific Background
- Implementations
- Going Parallel
- Results
- Conclusion
Scientific Background
Definition

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 $z = 0$. If we denote the Mandelbrot set by $M$, then by repeatedly applying the quadratic map

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

$\forall$ complex number $c$, we have $c \in M \iff \limsup_{n \to \infty} |z_{n+1}| \leq 2$. 
Generating Mandelbrot set images

- Sample complex numbers and iterate this function for each point $c$. If the function goes to infinity, the point belongs to the set.
- Each sampled point can be mapped on a 2D plane:
  - Treat its real and imaginary parts as image coordinates $(x + yi)$
  - Color this pixel according to how quickly $z_n^2 + c$ diverges

![Mandelbrot set drawings generated with our palettes.](image)

(a) Red-shaded  
(b) Smooth, blue-shaded

**Figure 1:** Mandelbrot set drawings generated with our palettes.
Algorithm 1 Escape Time algorithm

1: for each pixel (Px, Py) on the screen do
2:    $x_0 \leftarrow$ scaled x coordinate of pixel
3:    $y_0 \leftarrow$ scaled y coordinate of pixel
4:    $x \leftarrow 0.0$
5:    $y \leftarrow 0.0$
6:    iteration $\leftarrow 0$
7: while $(x \times x + y \times y < 2 \times 2 \text{ AND } iteration < \text{max\_iteration})$ do
8:     $x_{temp} \leftarrow x \times x - y \times y + x_0$
9:     $y \leftarrow 2 \times x \times y + y_0$
10:    $x \leftarrow x_{temp}$
11:    iteration $\leftarrow$ iteration + 1
12:    color $\leftarrow$ palette[iteration]
13:    plot($Px, Py, color$)

Where $z = x + iy$, $c = x_0 + iy_0$, $x = \text{Re}(z^2 + c)$ and $y = \text{Im}(z^2 + c)$.

**Embarrassingly parallel problem:** each pixel independent of any other.
Implementations
Implementations overview

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 application is coded in C.

Code fully debugged and profiled using gdb and gprof. Valgrind used to make sure no memory leaks are possible.
Serial Implementations (1)

- naive.
- **opti1**: Exploit symmetry with respect to the $x$ axis.
- **opti2**: Skip calculations for the points lying within the cardioid or in the period-2 bulb ($\approx 34\%$ of the total area).
- **opti3**: Finer-grained optimizations: reduce to 3 multiplications per loop (minimum) & use fast array indexing in plotting.

![Figure 2: Periods of hyperbolic components.](image)

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Execution time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00</td>
<td>422.91</td>
</tr>
<tr>
<td>-01</td>
<td>230.99</td>
</tr>
<tr>
<td>-02</td>
<td>208.96</td>
</tr>
<tr>
<td>-03</td>
<td>208.49</td>
</tr>
<tr>
<td>-03 -ftree-vectorize</td>
<td>205.41</td>
</tr>
</tbody>
</table>

**Table 1**: Execution times of **opti3** (368.64M pixels, max_iteration=10,000) as a function of the optimization level.
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).
Going Parallel
CUDA Implementations (1)

- **opti3**: Parallelize escape time evaluation only.
  - Colors computed serially
  - 3 matrices sent on the PCIe bus

- **opti4**: All workload carried out by the GPU.
  - Only exchange RGB values of the bottom part on the PCIe bus
  - Use *Constant memory* to store constant values on the GPU
  - Code split into serial (*.c) & parallel (*.cu) parts; needs C wrapper

- **opti5**: Single-precision implementation of opti4.
  - Maximize instruction throughput
  - Cannot distinguish between double- and single-precision images

Application profiled with *nvprof*: compute-bound.
Figure 4: 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.
CUDA block size tuning for opti5

Block size:

- hyperparameter leading to performance improvements, increasing occupancy \( \left( \frac{\text{active warps}}{\text{maximum active warps}} \right) \)
- It depends on both the problem and the specifics of the GPU

<table>
<thead>
<tr>
<th>Block size</th>
<th>Occupancy [%]</th>
<th>Execution time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2560x1440</td>
<td>25600x14400</td>
</tr>
<tr>
<td>32x1</td>
<td>18.05</td>
<td>18.81</td>
</tr>
<tr>
<td>64x1</td>
<td>25.41</td>
<td>32.65</td>
</tr>
<tr>
<td>128x1</td>
<td>31.82</td>
<td>51.52</td>
</tr>
<tr>
<td>256x1</td>
<td><strong>36.15</strong></td>
<td><strong>58.80</strong></td>
</tr>
<tr>
<td>512x1</td>
<td>28.76</td>
<td>52.37</td>
</tr>
<tr>
<td>1024x1</td>
<td>8.39</td>
<td>25.50</td>
</tr>
<tr>
<td>8x8</td>
<td>19.04</td>
<td>20.12</td>
</tr>
<tr>
<td>16x16</td>
<td>26.91</td>
<td>25.93</td>
</tr>
<tr>
<td>32x32</td>
<td>5.91</td>
<td>7.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block size</th>
<th>Occupancy [%]</th>
<th>Execution time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2560x1440</td>
<td>25600x14400</td>
</tr>
<tr>
<td>32x1</td>
<td>2.53</td>
<td>79.61</td>
</tr>
<tr>
<td>64x1</td>
<td>2.29</td>
<td>50.32</td>
</tr>
<tr>
<td>128x1</td>
<td>2.10</td>
<td>37.52</td>
</tr>
<tr>
<td>256x1</td>
<td><strong>2.04</strong></td>
<td><strong>35.91</strong></td>
</tr>
<tr>
<td>512x1</td>
<td>2.17</td>
<td>37.69</td>
</tr>
<tr>
<td>1024x1</td>
<td>5.65</td>
<td>90.60</td>
</tr>
<tr>
<td>8x8</td>
<td>4.09</td>
<td>237.72</td>
</tr>
<tr>
<td>16x16</td>
<td>2.45</td>
<td>97.12</td>
</tr>
<tr>
<td>32x32</td>
<td>7.48</td>
<td>270.24</td>
</tr>
</tbody>
</table>

**Table 2:** Occupancy and execution time for two image sizes as the block size varies for max_iteration=10,000. Evaluated using `nvprof`. 
Hybrid Implementations (1)

Distribute the computations over different CPUs and GPUs with MPI.

- Image split to the 2 processors in a node, each uses 1 GPU.
  - Non-blocking, point-to-point communications where rank 0-processor receives the image parts out-of-order \(^1\)
  - Parallel I/O
- Image split to the 2 processors in a node, each uses 2 GPUs.
  - Highest performance: all the CUDA functions are asynchronous \(^2\)
  - Non-blocking, point-to-point communications where rank 0-processor receives the image parts out-of-order \(^1\)
  - Parallel I/O

**Figure 5:** Image split to CPUs (orange) and GPUs (orange or green).

\(^1\)No MPI_Barrier() used.
\(^2\)No __syncthreads() used.
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 $256 \times 1$. 
Results
Strong Scaling

**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 (368.64M-pixel image, max iteration=10 × 10^6) for the different versions.
Figure 8: Weak scaling as a function of the problem size that can be solved in the same amount of time (76 s) by the different versions.
Conclusion
Conclusion & Future Work

Conclusion

- Implement and optimize serial, CUDA and CUDA+MPI applications
- Gain experience with CUDA and MPI in a supercomputer
- Acquire familiarity with profiling tools
- Investigate different technologies
  - Non-blocking communications and parallel I/O in MPI
  - Asynchronous functions in CUDA

Future Work

- Experiment with 2D grids
- Investigate the impact of external libraries
- Explore different algorithms
Conclusion & Future Work

Conclusion

- Implement and optimize serial, CUDA and CUDA+MPI applications
- Gain experience with CUDA and MPI in a supercomputer
- Acquire familiarity with profiling tools
- Investigate different technologies
  - Non-blocking communications and parallel I/O in MPI
  - Asynchronous functions in CUDA

Future Work

- Experiment with 2D grids
- Investigate the impact of external libraries
- Explore different algorithms

Thank you