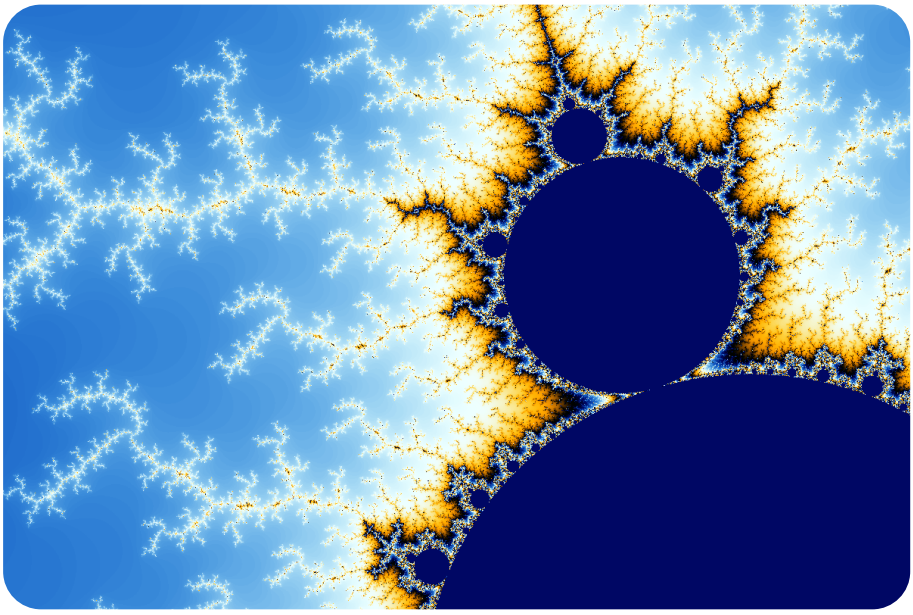
Computation-bound Benchmark: Mandelbrot Escape Time Algorithm in CUDA

Homework assignment for CS 7331

Dillon Lohr
Texas State University
Spring 2019



Code available on GitHub

Assignment

Write a synthetic benchmark that is compute-bound on system X.
Demonstrate using hardware performance counters.

1 Introduction

According to Wikipedia, "[t]he 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 $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value." Technical details aside, representations of the Mandelbrot set can produce very pleasing visuals, such as the image on the title page. One simple, naive approach to visualizing the Mandelbrot set is the "escape time" algorithm.

In brief, the escape time algorithm performs repeating computations for each pixel position, and a color for the pixel is chosen based on how many repetitions are needed to reject or fail to reject that point as being a member of the Mandelbrot set.

2 Benchmark Description

The benchmark works as follows.

1. Process command-line arguments
2. Allocate memory for the host and device
3. Launch the kernel with one thread per pixel, deciding between single- or double-precision calculations based on the precision needed
4. Wait for the kernel to finish
5. Save the generated image(s) if desired (requires copying the results from the device's memory to the host's memory)

The kernel repeats the member checking computations up to 65,536 times before conceding that the point is probably a member of the Mandelbrot set. The number of times to magnify the set must be specified by the user at runtime, along with the desired width of the generated image(s). The height of the image is automatically set to two-thirds the width to maintain the desired aspect ratio. One image is generated for each magnification level, with deeper magnifications requiring more floating-point precision to accurately compute the image. Due to this increasing precision requirement, the benchmark decides between using single- or double-precision calculations based on the final magnification level.

Some optional parameters can also be specified. First, the user can decide if the generated image(s) should be saved (disabled by default). Second, a starting magnification level can be provided. This helps if the work needs to be split up across multiple program executions due to memory size limitations when working with very large widths and/or dozens of magnification levels. Finally, the magnification factor can be passed in as a command-line argument, allowing the user to specify how quickly (or slowly) deeper magnifications can be reached.

For the purposes of this report, the benchmark was run on the Discovery machine with the following settings and constants (a low workload was necessary for metrics extraction, because `nvprof` kept overflowing):

setting	value for metrics extraction
`depths`	16
`width`	600
`save_output`	0
`depth_start`	0
`zoom_factor`	0.9
`MAX_ITERATIONS`	65536
`THREADS_PER_BLOCK`	256

Lastly, the program was compiled using the command below.

```
nvcc -std=c++11 -O3 -arch=sm_60 mandelbrot.cu -o mandelbrot
```

3 Performance Metrics

The following command was used to extract the performance metrics, and the output `.prof` file was examined using the NVIDIA Visual Profiler:

```
nvprof --kernels mandelbrotKernel --analysis-metrics \
    -o analysis.prof ./mandelbrot 16 600
```

category	metric	value
memory	Load/Store Instructions	3,840,000
compute	FP Instructions (Single)	367,794,220,992
compute	Integer Instructions	92,232,979,306
compute	Control-Flow Instructions	45,960,933,171
memory	System Memory Utilization	Low (1)
memory	Device Memory Utilization	Low (1)
memory	Local Memory Overhead	9.2%
memory	Issue Stall Reasons (Memory Throttle)	0%
memory	Issue Stall Reasons (Data Request)	0%
compute	Issue Stall Reasons (Execution Dependency)	19.7%
compute	Warp Execution Efficiency	79.9%
compute	Warp Non-Predicated Execution Efficiency	79.8%
compute	FP Ops (SP Add)	183,816,486,496
compute	FP Ops (SP FMA)	46,037,533,624
compute	FP Ops (SP Mul)	91,914,011,248
compute	FP Ops (SP Special)	34,544,000

4 Analysis

We can see from the metrics table that there is low memory utilization, relatively few load/store instructions, and no memory-related stalls. We can also see that there are billions of floating-point operations being performed, and the warps are not being used optimally (only 80% efficiency). Therefore, it is clear that this problem is compute-bound rather than memory-bound.

5 More Output Images