PA03: Mandelbrot Image
CS791v: Parallel Computing

Terence Henriod

February 12, 2015

Abstract

The Mandelbrot set is a well known fractal among math enthusiasts. Computation of Mandelbrot sets/images is known to programmers to be an "embarrassingly parallel" task. The task is somewhat compute heavy, and the only communication required (if any) is to collect the results of the computaion. In this assignment, we explore computation of Mandelbrot images on the GPU.

1 Introduction

Computation of Mandelbrot images is “embarassingly parallel” due to the fact that pixel values can be generated without communication, they can be computed independent of one another, and the only communication required through the whole process is to collect the computed pixel values in one location once the computation completes.

2 Theory

2.1 Computing Pixel Values

Computing a pixel value follows the flowing equation: $z_k = z_{k-1}^2 + c$ where z is the pixel value, and c is the coordinate of the pixel in the complex plane. This formula is computed iteratively until either a maximum number of iterations has been reached (of note is the number 1024 for this exercise) or until z begins to converge to zero or diverge to ∞ (which is know to happen if z ever reaches 2).

2.2 Sequential Algorithm

This algorithm is straightforward. The simple use of a for loop (or pair of nested for loops) to visit the storage location for each pixel’s value is used, and at each location the pixel’s value is located. The value of the pixel is dependent on its coordinates in the image, but these coordinates can be computed easily from the overall array index and the width of the image. That is, the coordinates x, y can be computed from the main data array index i and the image width w by:

$$x = i/w$$

$$y = i\%w$$

2.3 Parallel GPU Algorithm

The GPU algorithm is also straightforward. Each GPU thread uses the same algorithm with one small difference: instead of incrementing the location of the next pixel to compute by one, each GPU thread strides across the data array in steps of the number of total threads.

I was curious about doing a version where a sort of “work queue” would be implemented using an integer in the shared block memory indicating the next pixel to be computed’s coordinate(s) and atomic operations to increment the shared value. The premise of this idea was that different pixels take different amounts of time to compute, so we could better balance the load on the threads using the work queue. Unfortunately, I ran out of time/motivation to experiment with this idea. It would indeed require more effort to implement, but I think that for very large images it might have its merits.

3 Results

The performance results of the vector reduction implementation are listed here.

3.1 Information on the GPU device used

Attribute	Value
Device Name	NVS 5400M
Cuda Version	2.1
Multiprocessors	2
Clock Rate	950 mHz
Total Global Memory	1073 MB
Warp Size	32
Max Threads/Block	1024
Max Threads-Dim	1024 x 1024 x 64
Max Grid Size	65535 x 65535 x 65535
SharedMem/Block	49 KB

3.2 Performance Graphs

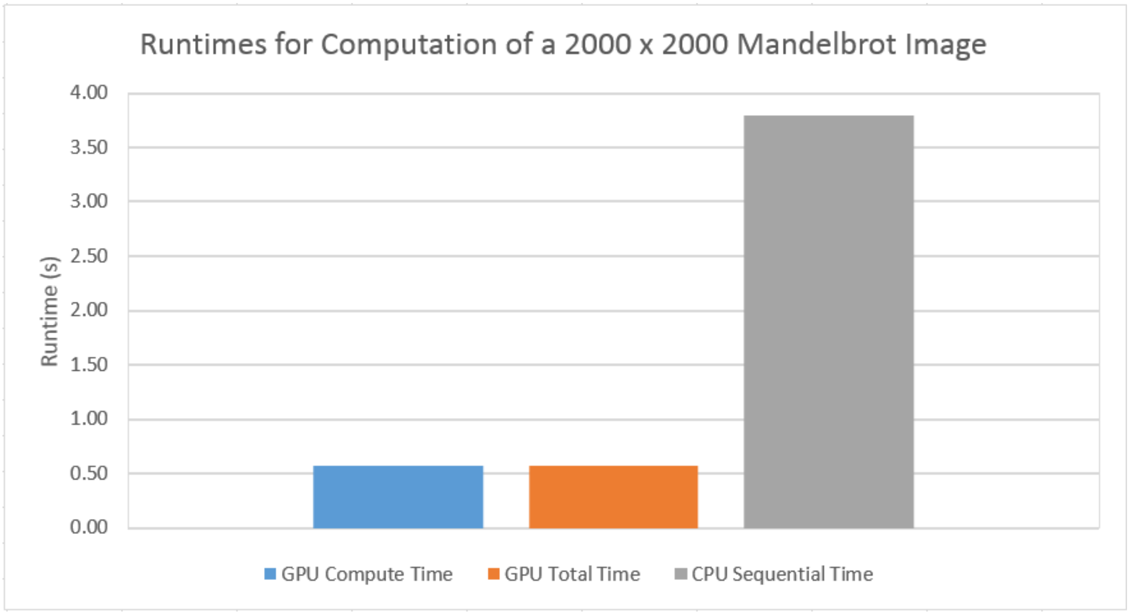

Figure 1: A comparison of the runtimes for computing a 2000x2000 Mandelbrot image using 1024 iterations at most for each pixel.

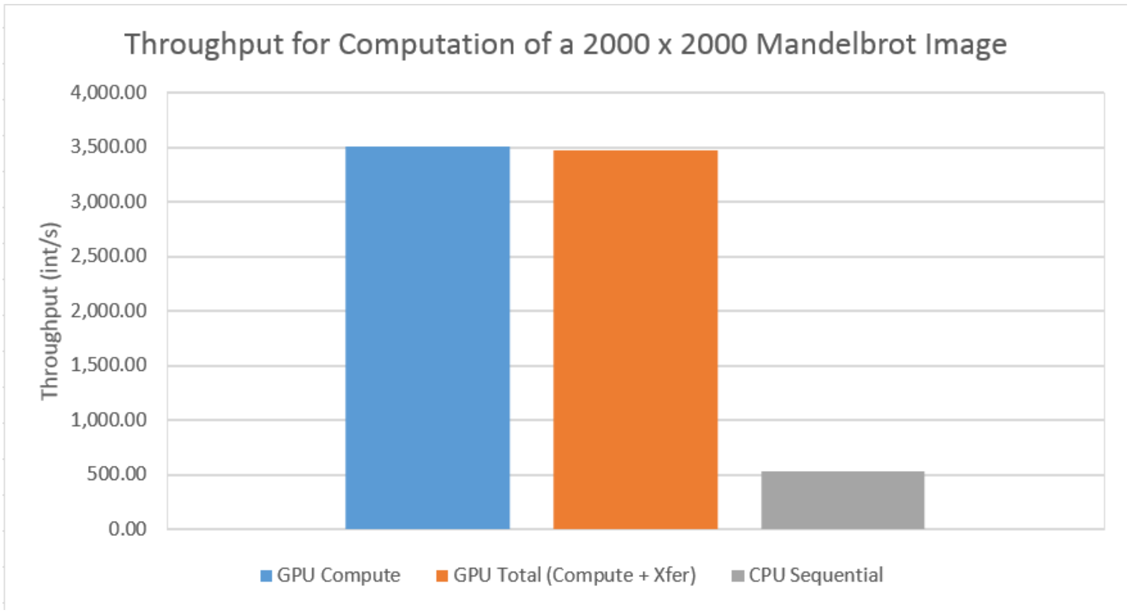

Figure 2: A comparison of the throughput in integer pixel values produced per second.

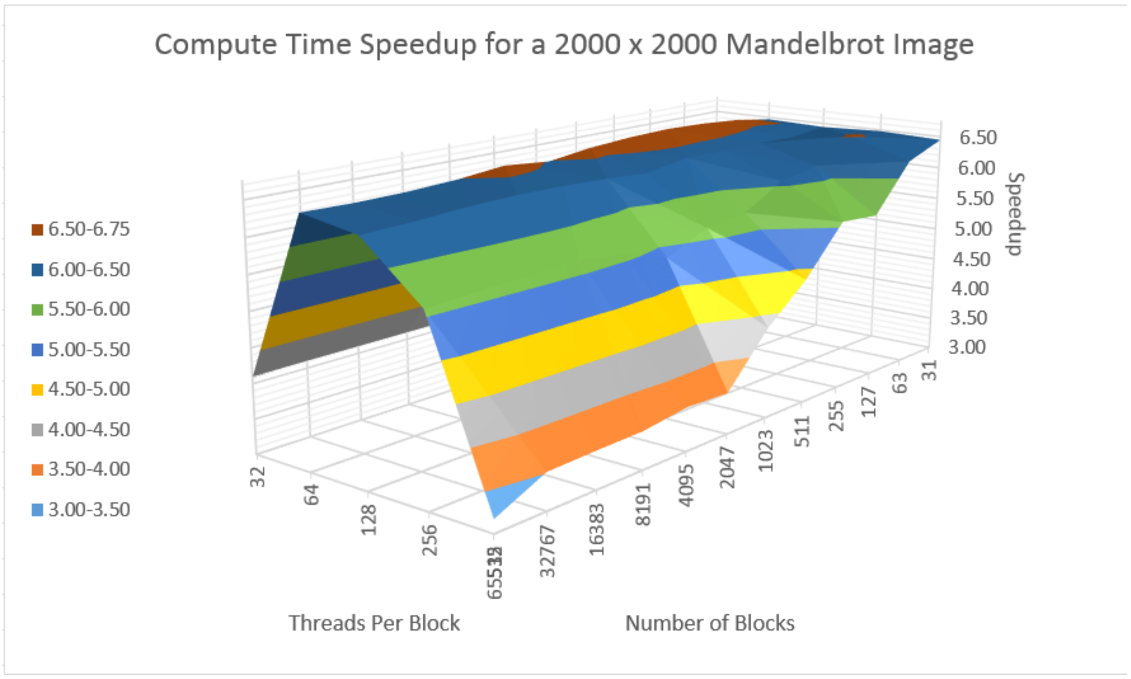

Figure 3: The speedup achieved. Note the peak in the surface at 64 threads for all block sizes.

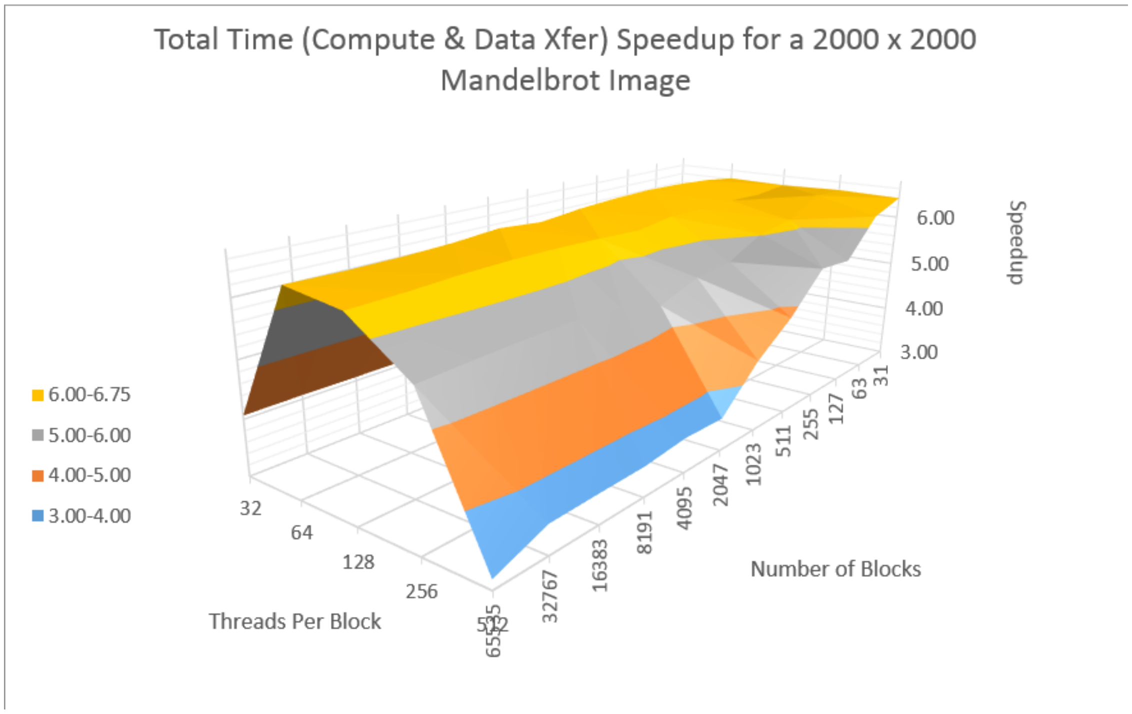

Figure 4: The speedup achieved when also considering the time to copy the computed data from the GPU to the Host.

4 Discussion

4.1 Low Speedups

The speedups were not as high as I had thought they would be. I believe that this is likely due to my use of a weak GPU - specifically the one that is in my laptop. I have little reason to believe that the lack

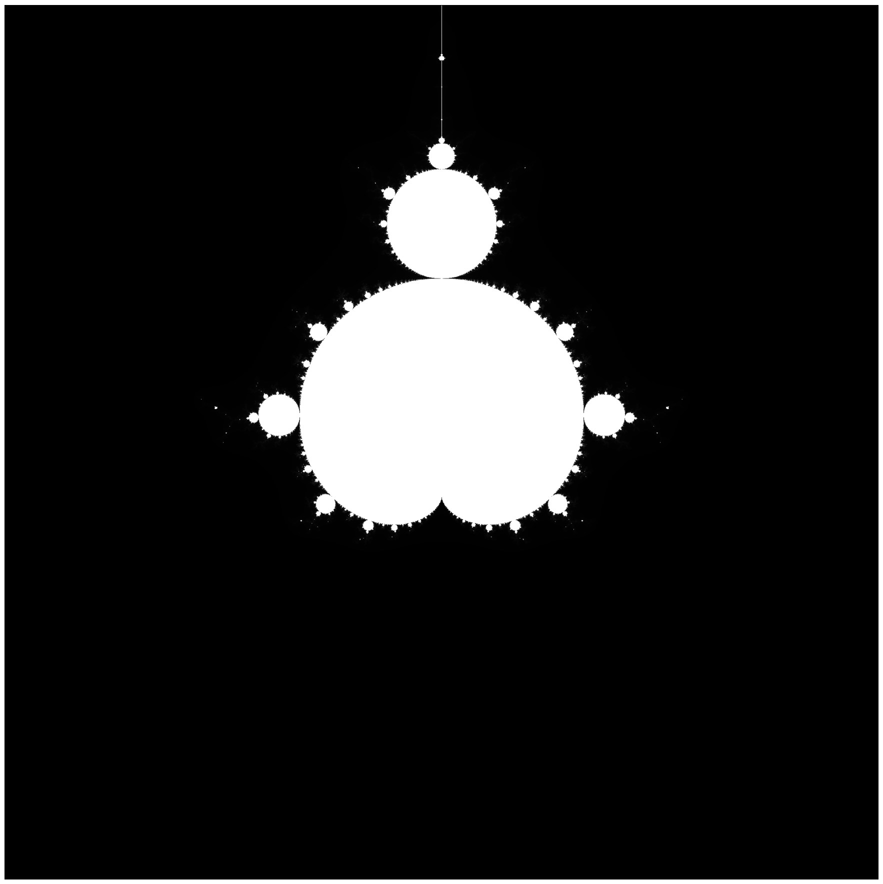

Figure 5: A Mandelbrot image for verification.

of speedup was due to a mismatch of algorithms because the sequential and the parallel algorithms are so similar, literally the only difference is the increment factor in the for loop. It is not impressive, but I was able to write CUDA code and do the explorations for the assignment.

4.2 The Right Number of Threads/Blocks (or The Right "Stride")

The highest speedup (for the 2000 x 2000 image) was observed using 255 blocks 64 and threads per block to produce a speedup of approximately 6.64. Looking at the speedup surface graphs do seem to indicate that the number of threads per block was more important than the number of total blocks used. I believe that this is likely because Mandelbrot computation has a high computation to memory access ratio. It is likely that since the threads only need to make 1 global memory access at the end of their computation, so it is better to keep the number of threads low so that they can spend their time performing the computations.

4.3 Lack of Transfer Time

Unlike with many other processes that are parallelized on the GPU, memory transfer time was almost negligible for this task. Only one transfer needs to occur since GPU threads ca figure everything about a pixel out based on which index they are processing

5 Issues

5.1 Weak Hardware Setup

The GPU used was a medium caliber laptop card. This resulted in limited performance, memory sizes, thread/block counts, etc. In addition, the Windows operating system was used, meaning that all GPU operations needed to occur within 2 seconds. This limited My data collection from using anything less than 32 threads per block and be able to have consistent success (16 threads worked often, but not often enough). However, I believe enough data was gathered to be able to comprehensively address the topic.