Marshall Mattingly
CSCI 532 - Spring 2016

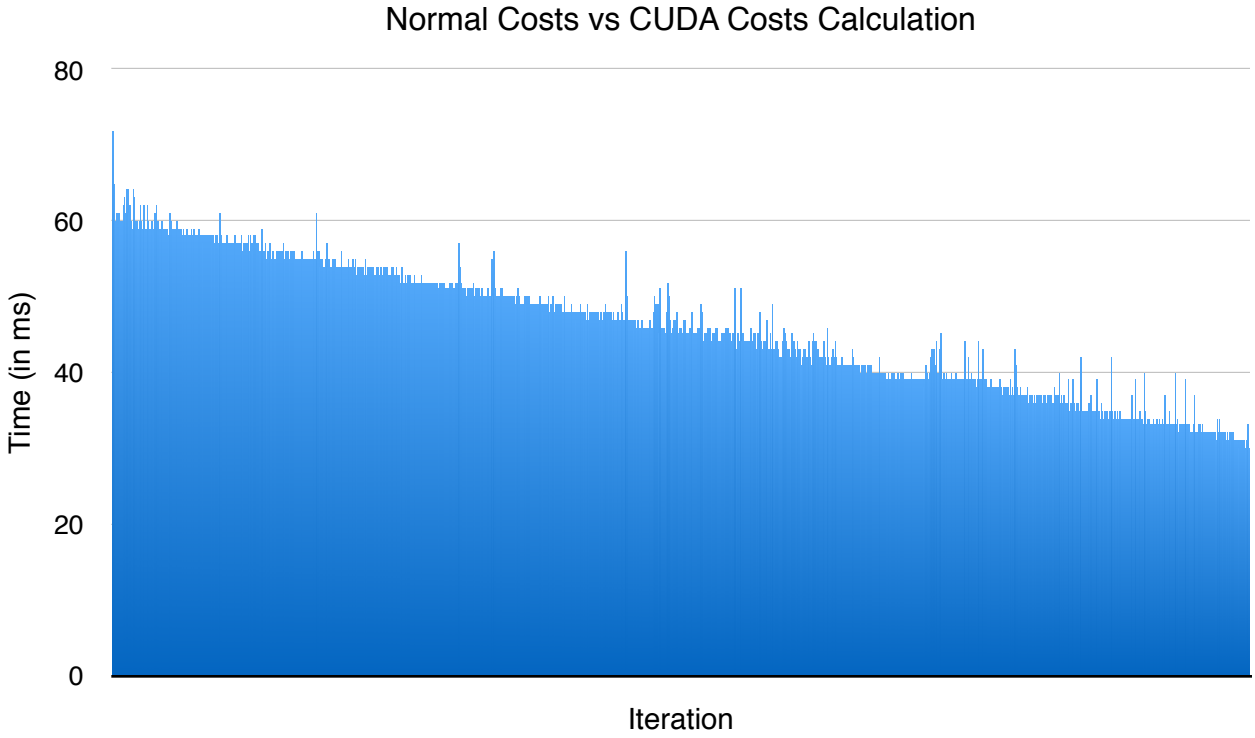
Homework 4 Report

The seam carving algorithm analyses an image to determine a seam — an unbroken line from top top bottom that intersects a row once — of least importance to remove. The user is able to decide how many vertical seams to remove to shrink the width of an image down by the specified pixel amount.

To calculate the individual seams, the seam carving algorithm goes row by row, attempting to approximate the importance of each individual pixel by adding its value to the pixel directly above and the pixels to the left and right to the one directly above. This determines the cost of each pixel within the image, as well as the direction of the flow for the given seam.

This cost calculation lends itself extremely well to parallelization. The basic premiss is that a GPU, able to process 1024 threads simultaneously, can dedicate a thread to each column in the image. Each thread then updates the pixel costs, row by row, in order, reading in the next row as they go and storing it in shared memory — a technique called tiling — to minimize costly main memory calls.

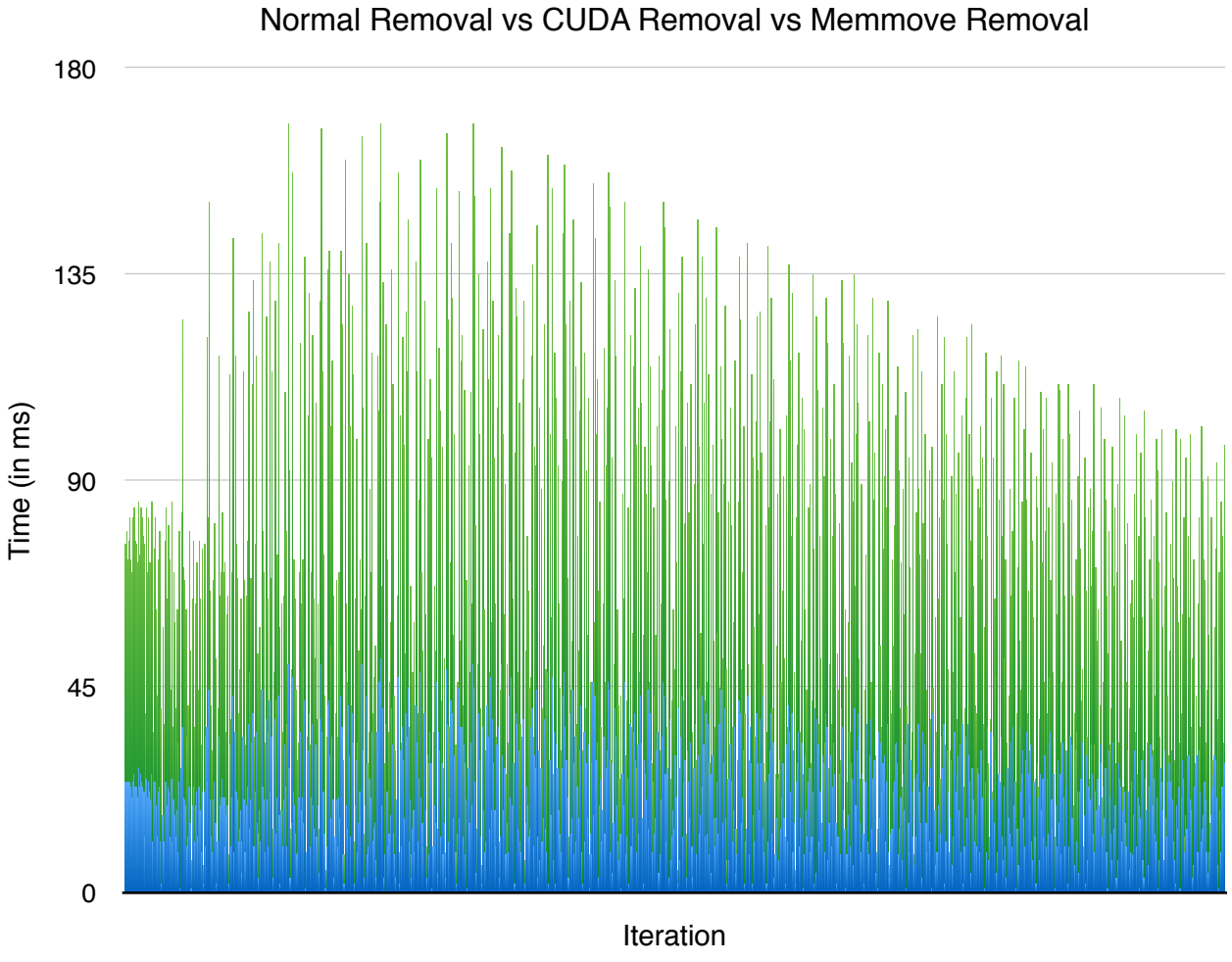
We can see in the following graph just how much gain we get from using CUDA to calculate the costs. The Normal Cost method (in light blue), using basic CPU iteration, starts out around 60 ms for a 1920x1080 image of a scene from Minecraft, and linearly decreases as remove the amount of iterations required — as we remove seams, we remove a single column from the calculation. However, the CUDA Cost method (in sea green) sticks almost perfectly to a 9 ms mark. The CUDA method likely doesn't decrease because we still iterate over the entire image,



we just do a “no call” for x values outside the current bounds by calling continue without the for loop.
After the costs are computed and stored in an array, the seam carving algorithm needs to determine the correct seam starting point and traverse the seam, storing it in an array of the same height as the image. This particular section doesn’t lend itself well to parallelization as each point needs to know the computed value of the previous point, which isn’t available until the previous point is calculated. This linear time requirement makes distributing the algorithm difficult and impractical, especially since the algorithm takes negligible time — 1-2 ms on average.

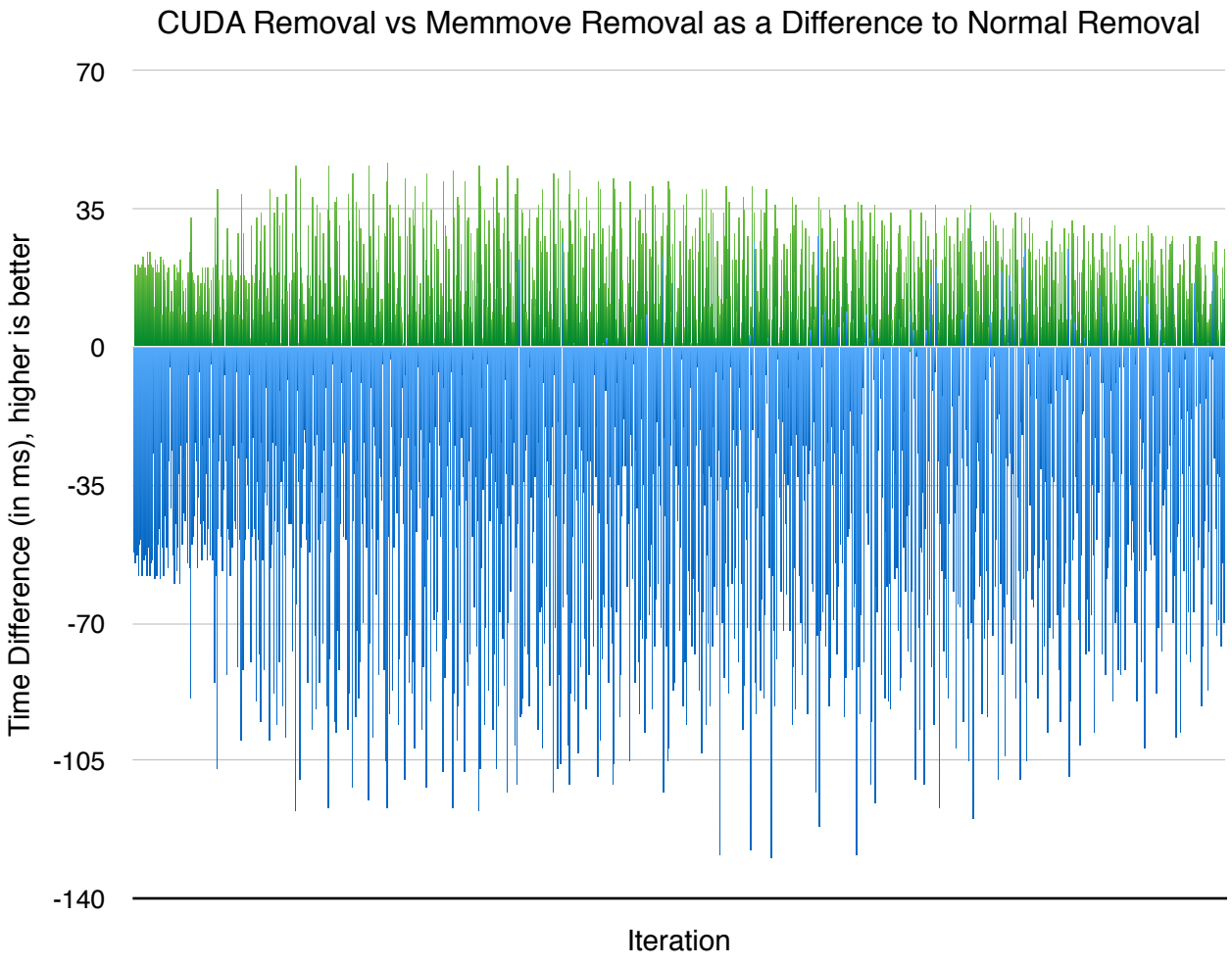
The final part of the seam carving algorithm is to actually remove the seam from the image. To do this, we follow the seam determined in the previous step, and shift every value to the right, in the row, to the left by one. Then we set the final value to 0. We’ll compare a few methods in the following graph.

The Normal Removal method (in blue) iterates over the image and value arrays, updating that array as it goes before setting the final value to 0. The Normal Removal method takes between 25 and 35 ms to compute, on average. The CUDA Removal method (in blue) using the same logic while distributing the workload over 1024 threads, with each thread responsible for a row. Surprisingly, the CUDA Removal method took between 90 and 120 ms to compute, on average, significantly worse than the Normal Removal method. It’s likely that the CPU iteration is able to be optimized by the compiler to require significantly less memory overhead, while the GPU iteration has less compiler optimization and his hitting global device memory at a much higher rate, significantly reducing the throughput.



Optimizations to the CUDA Removal method were attempted, but unsuccessful. However, after releasing that the data in each row is stored directly after each other in memory, a Memmove Removal (in dark green at the bottom) method was created. The Memmove Removal method works by calling memmove to move the required amount of bytes to the right of the seam into the address of the seam, effectively removing the need to iterate over the row entirely. Even with the additional cudaMemcpy to update the vals array on the GPU, the Memmove Removal method takes a mere 3 ms no matter the size of the array, significantly faster than either other method.

The difference is highlighted even more in the following graph, showing just how much better the Memmove Removal method is relative to the Normal Removal method compared to the CUDA Removal method.



This project shows just how important it is to consider different computational methods for individual pieces of code, and how benchmarking those methods can lead to some important insights. By using the CUDA Costs method, calculating the seam on the CPU, and using the Memmove Method, a 1920x1080 image from Minecraft was reduced down to a 920x1080 (1000 seams removed) image in just 18 s, compared to 70 s required by using the CPU alone.