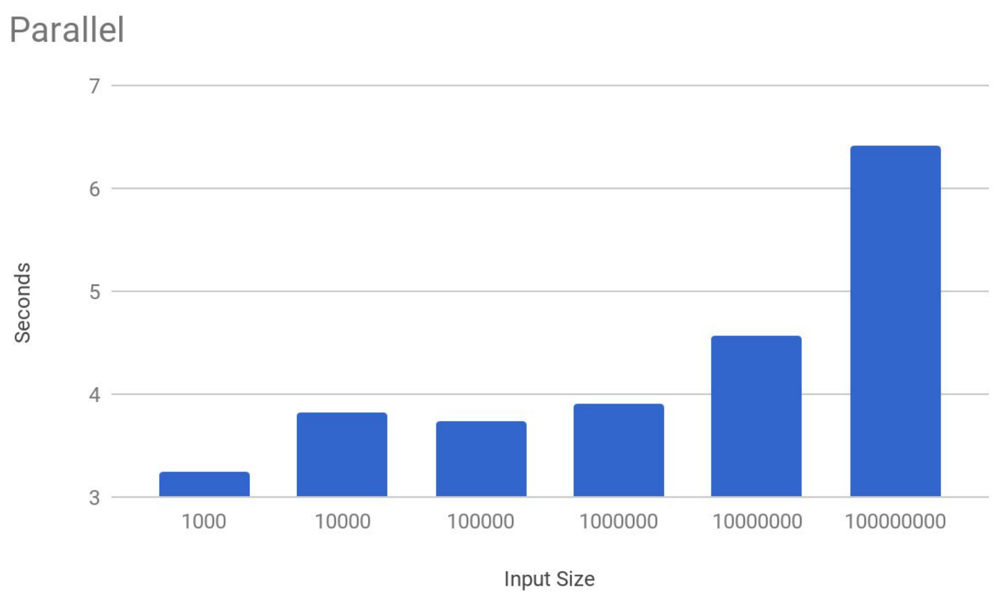
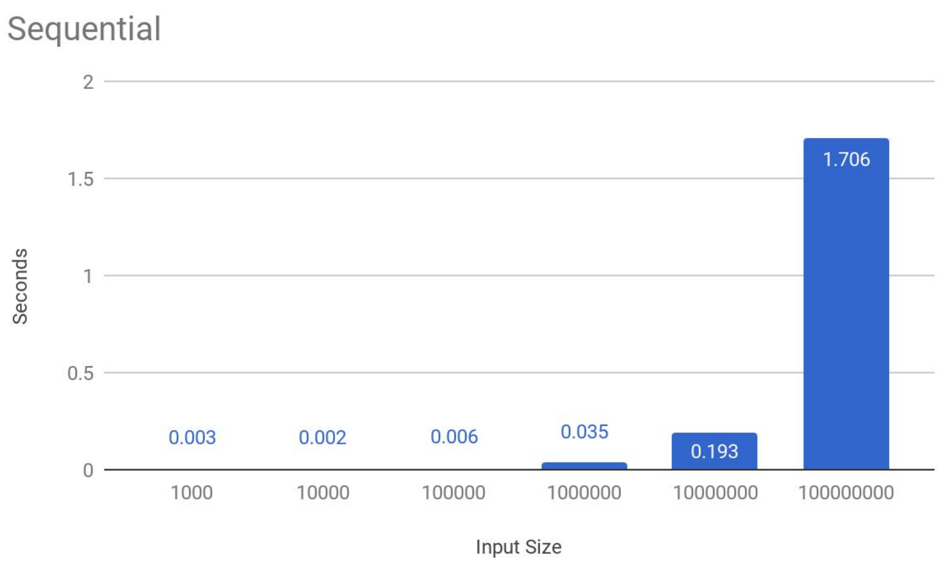
Andrew Huang
amh877
Parallel Computing Spring 2018

I used CUDA2 to compile my program

1. I used the max block size of 1024 as the thread count per block because that is the maximum supported by the 3.x architecture. This way I can full take advantage of shared memory. I then divided the work evenly among all of the blocks, using modulo on the parts array that may overflow and not divide evenly and extending the array as appropriate.
2. I compiled my code with `nvcc -arch=compute_30 -code=sm_30 -o maxgpu maxgpu.cu`
3.





Andrew Huang
amh877
Parallel Computing Spring 2018

4. The runtime of the parallel code is adversely affected by copying the memory to the GPU, and then accessing the shared memory. Even though I used the reduction algorithm, I experienced a runtime cost from cuda. However, even though the runtime is worse, we can see that from the speedup grows as the input size increases, suggesting that with larger input sizes (given that the C environment would allow that large of an array) would benefit from using the cuda version. The synchronization points of the Cuda program also hindered its performance because when finding the max in it's shared memory, we have to make sure that all of the threads have finished up to that point. Additionally, we had to copy the entire array over to the gpu, and then reduce the maximum, and copy the result array back to the cpu. However, in the native C code there is no such branch condition, and it can continue, also without the bottleneck of copying memory.