Alperen Karaoglu ak4410 N130120

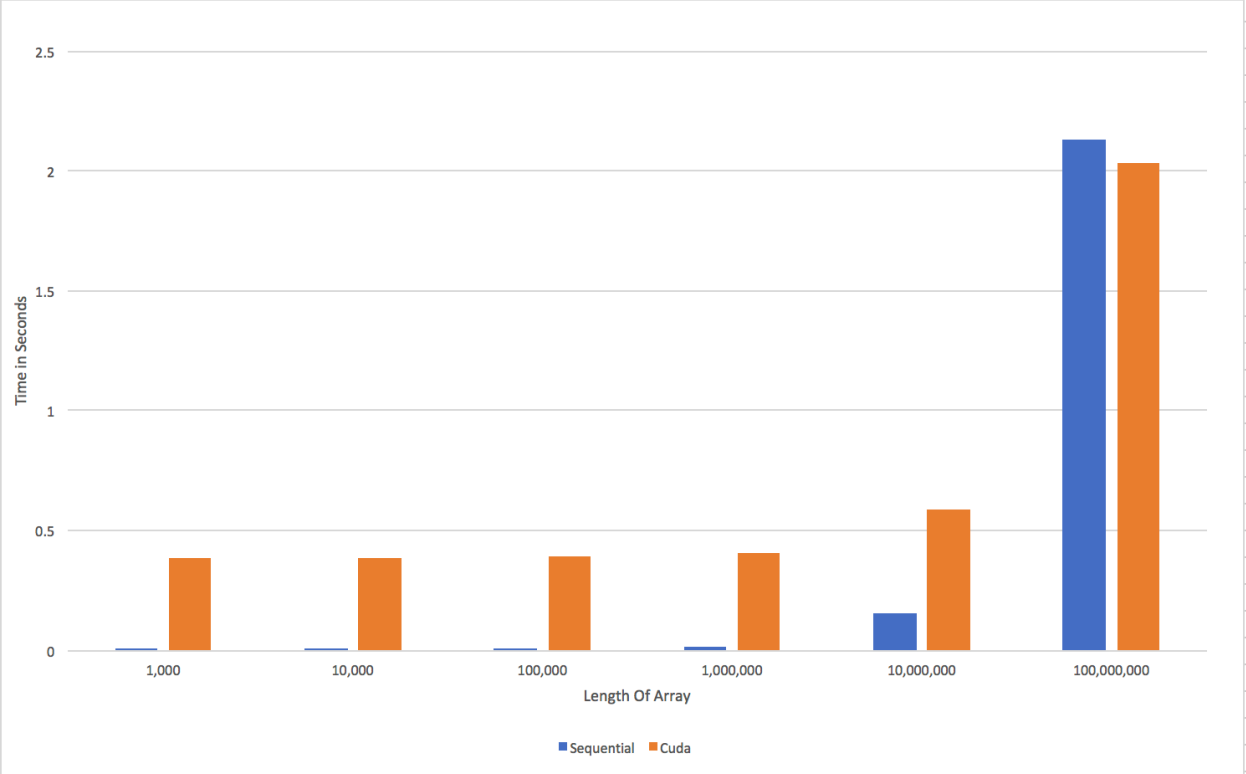
Parallel Computing: Lab3 Report

Using cuda1 to run the experiments:

1. Given the array size, pick the maximum number of threads per block such that we use one thread for each element of the array. Once the number of threads we need is determined, pick the minimum amount of blocks necessary to fit all those threads. Grid dimensions are 1 * number of blocks. With this approach reduction becomes simple, using the maximum amount of threads we can, we reduce the array the minimum number of blocks. If after the first iteration we don't have a single block and thus a single value, we repeat the process until single value reduction is achieved. By picking the minimum amount of blocks that can be used we also minimize the amount of times we have to load from the device's global memory to every block's shared memory.

2. nvcc –arch=compute_30 –code-sm_30 maxgpu.c

3.



4. Because the sequential version compares every value in the array one by one, its execution time increases exponentially with the array size. For data sets that are small, the cuda execution time is slower than the sequential code because of the time it takes to load from host to kernel

and more importantly the time it takes to load from global memory to shared memory and then to global memory again in the device. Thus for small data sets the benefit of using multiple threads to reduce the array is lost due to the memory overhead. When the data set is large enough, in this case 100,000,000, we finally see speedup compared to the sequential version. The advantage of using multiple threads to reduce the array over and over again to a single value becomes more apparent, especially since we don't have to transfer between host and device for each iteration, we just need to do it once. The overhead from loading between the global memory of the device and the shared memory of each block becomes less significant due to the bigger advantage from using multiple threads on a larger data set.