Jessica Donahue
Parallel Computing
4/6/16

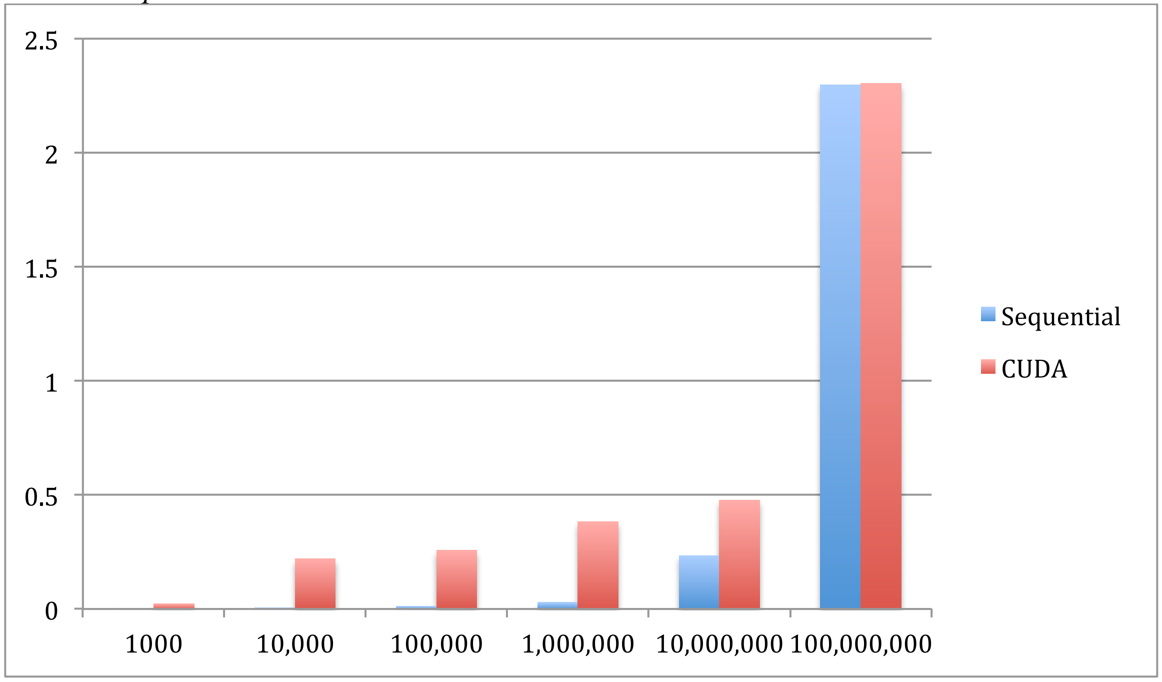
Lab 3 Report

1. *Block and grid size dimensions:*
I simply maximized the numbers of blocks and threads for each array size. I found the maximum number of threads per block by using the *maxThreadsPerBlock* API and then figured out how many blocks were needed for that amount of threads. This way I am using all the threads available for each device. Therefore, the grid size dimensions are 1 by the number of blocks.

2. *Command line to compile the CUDA version:*

```
nvcc maxgpu.cu –o maxgpu
```

3. *Bar Graph:*



4. *Conclusions:*
From the bar graph it is clear that the runtime for both the sequential and CUDA version of the code increase with increasing array size, in an exponential pattern. The sequential version has to iterate through every single element of the array and make comparisons between the elements. Therefore, the code is inefficient for large numbers of data and this is why the sequential runtime greatly increases from 10 million to 100 million elements. Although, the CUDA version's runtime is significantly greater for 1,000 to 10 million elements, the runtime for the sequential and CUDA versions become similar for 100 million elements. The runtime for the CUDA version is greater than the sequential version for the smaller elements due to the expensive operation of accessing global memory. In the kernel function, every thread must access global memory, which is located outside of the block that the thread resides in. Thus, although the CUDA program is experiencing parallelism, as the threads in the independent blocks compare

elements at the same time, we are not experiencing performance improvements due to the constant global memory accessing by the threads. Therefore, both versions of the code prove to not scale for large sets of data. The sequential version becomes inefficient because every element has to be iterated through sequentially and the CUDA version becomes inefficient due to the global memory access time.