cuda2 (Device Number 1) was used during experimentation.

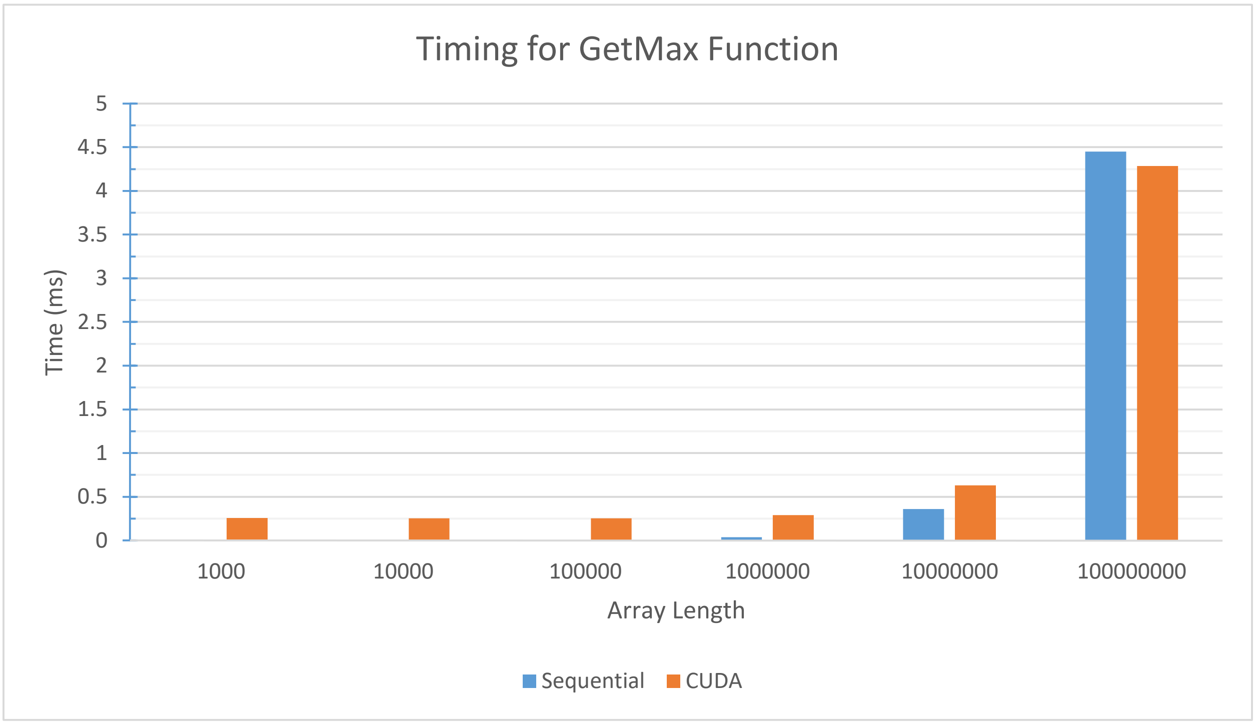
1. The grid and blocks were each chosen to be 1D to match the geometry of the problem which was simply a 1D array of numbers. Since altering the dimensions does not add any speed-up or slow-down either, it made the most sense for each thread in a block to also be accessing elements in a linear 1D array.

   The block size was chosen to be 256. The max threads per block was found to be 1024, so this was within hardware limitations. 256 is also a multiple of the warp size which was found to be 32. This would allow the blocks to be partitioned into 8 warps each. In cases where the threads in a warp had to perform memory accesses, there were 7 other warps that could begin execution. It was also found that a maximum of 2048 threads could be executed in a multiprocessor. This means that 8 blocks could be assigned to a multiprocessor. This would allow ample blocks and warps to be set to any SM in the case where global memory accesses occur. When this happened, another block/warp could begin execution in that SM.

   The grid size was chosen to be the ceil(LENGTH_OF_ARRAY/BLOCK_SIZE). This was chosen so that as many blocks as possible would be assigned to SMs. The grid size continuously gets smaller with multiple divisions by BLOCK_SIZE to find possible max elements.

2.
   ```
   $ nvcc -arch=compute_35 -code=sm_35 -o progname progname.cu
   ```
   progname = compiled program name (I chose maxgpu)
   progname.cu = CUDA code file – maxgpu.cu in this case
   compute_35 and sm_35 were chosen because the GTX TITAN Z had a compute capability of 3.5. The architecture and code arguments would allow the program to utilize Kepler's features.

3.



4. For very small array lengths, the CUDA program will take significantly longer to execute than the sequential. This can be seen for array lengths up to 1,000,000. This is the case because host memory is slow and device is slow. An ample amount of time is taken to simply allocate the memory on the device and then transfer the contents of the array from the host to the device. For small sizes, it is faster to let the host compute the max element sequentially.

   It is not until 10,000,000 and 100,000,000 that a noticeable comparison is seen between how long the GetMax operation takes. However, a larger amount of data also means more time must be taken to transfer all of that data. The sequential code ONLY performs the operating of finding the max. The CUDA code has to perform memory transfer AND finding the max.

   We see that for very large data (100,000,000 long elements in this case), the use of the GPU is worth the time it takes to transfer data from host to the device. In this case, there is enough data and parallelism so that all of the SPs are being used by threads to find the maximum element.

   However, the CUDA version could be more optimized. It could be even faster if memory transfer was overlapped with computation as opposed to letting the entire array be copied to the device, then executing kernels to find the max (which is what I did).