Matt Kwong

Parallel Computing Spring 2016

Lab 3

All of my testing was done on cuda5. I included the line "cudaSetDevice(1);" in my main function.

1.

For the block size:

I wanted a multiple of 32 as the block size because of the warp size of 32. I checked the device properties by using cudaGetDeviceProperties() and checking the maxThreadsPerBlock value. I then decided to use 1024, which was the max, because I did not see any scalability issues or memory accessing issues that would affect the performance if I maximized the block size.

For the grid dimension/size:

To calculate, I took the ceiling of array_size /block_size to get the # of blocks I would need.

Overall, I am using a tree-like structure to find the max, so maximizing the block size reduces the # of blocks necessary, so it's akin to having ore children per parent, so this reduces the depth of the tree and should yield better performance.

2.

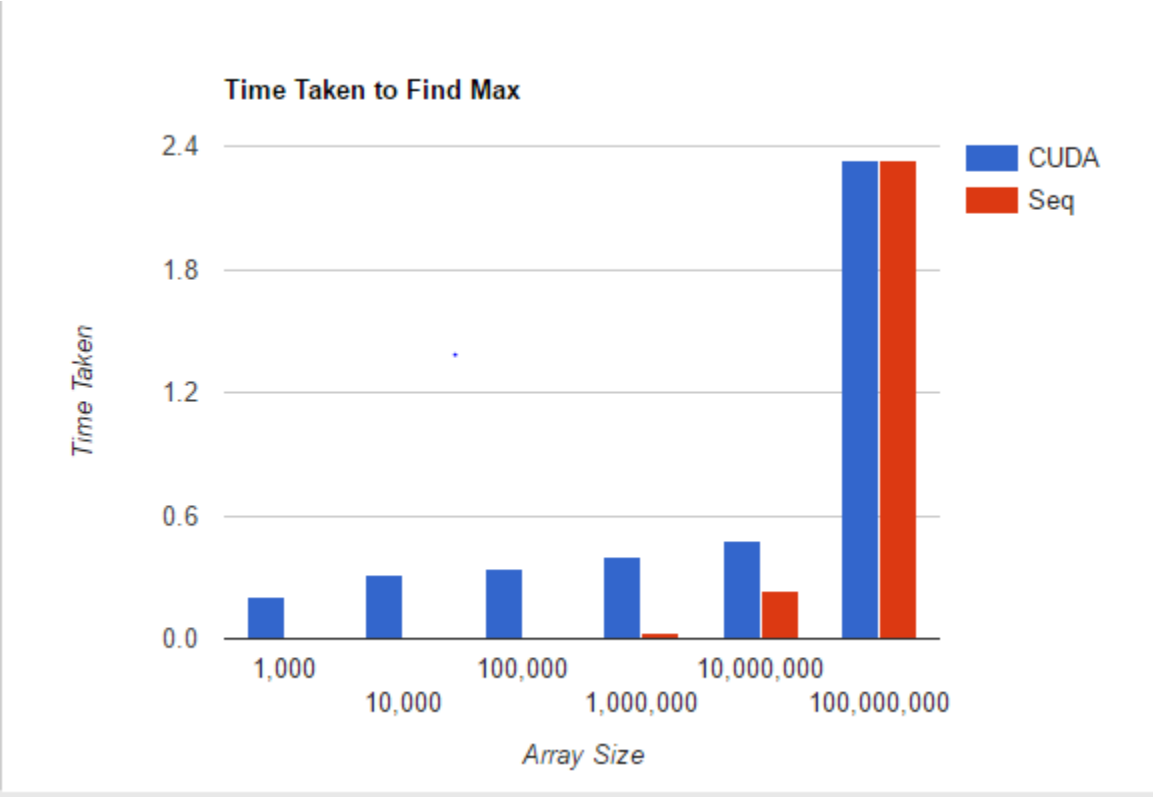
In addition to the commands stated on the website to set up CUDA, I used the command **nvcc maxgpu.cu** to compile the program. This makes the default output a file called **a.out**, but we can use the –o argument to specify the name of the output file. I then ran **time ./a.out *x*** where x is the size of the array I want to initialize. I tested this up to 100,000,000 as the lab instructs. I reported the real time reported from the time command.

3.

Time in seconds

	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
CUDA	.201	.311	.342	.402	.478	2.23
Sequential	.001	.002	.005	.029	.232	2.33

*times were taken as an average of 5 runs and the sequential is the source code given by the professor



4.

As we can see, both graphs seem to grow exponentially, but this is because the scale is increasing by a multiplicative factor of 10. The interesting element is that CUDA only beats the sequential code when the size of the array is at least around 100,000,000, and it barely beats it. Otherwise, we see that the sequential time is significantly better in every other tested array size.

The reason for this is the same as for the previous two labs: parallelizing code has a lot of overhead! When we want to do something simple and a small scale like finding the max value of an array under the size of 1,000,000, sequential code can do this almost instantly.

If we were to evaluate the shapes of the curve if scaled on a linearly growing x axis, the sequential curve would be linear, which can be seen if we look at the data and the last 3 times grow by around a factor of 10. This also makes sense because the sequential code essentially iterates over the array one-by-one to find the max value which is a linear search that would make the time taken also linear. The CUDA curve is logarithmic and this is illustrated in the data as the time grows faster in the smaller array sizes than the sequential code, but then grows slower than the sequential code somewhere between 1,000,000 and 10,000,000. My code also supports this because I am essentially make a tree structure to search the array as I find the maximum in subsets of the array and then the maximum of these subsets (similar to merge sort).

A tree's depth is a logarithmic function of the number of nodes, so that's why the time grows in logarithmic.

Overall, I believe that finding the max in subset of arrays and then finding the max of these was one of the better ways to use CUDA for this. I used merge sort to help me design this process, and since merge sort is one of the more efficient sorting algorithms, I believe this also makes my program have a good running time. I paid attention to designing my block and grid sizes/dimensions as well as designing my code to reduce cache misses with memory accesses by using memory coalescing, so overall, I think I made many of the major optimizations essential for CUDA performance.