Chris Jimenez
Dr. Mohamed Zahran
Parallel Computing
April 26, 2014

Programming Assignment 3 Report

Introduction

For this assignment a reduction algorithm was implemented to find the maximum element of an array of integers. The range of integers was between 1 and 100000. Firstly, a sequential version of the program was implemented in C. Then a CUDA version of the program was implemented that takes thread divergence in to considering and another CUDA version that does not. Lastly, another CUDA version was implemented that makes use of shared memory. This was done with an array of 8192 and 65536 elements. This totals to 8 source code files.

Algorithm

The implementation used an algorithm that split the work, the number of elements, in half and finds the max in each split. All the work is done on the array. This continues until there is nothing to split anymore, which eventually leads to the maximum value being at the first location of the given array(location 0). Given that the number of elements, both 8192 and 65536, are both powers of 2, the algorithm worked perfectly and symmetrically. Had the number of elements not have been a factor of 2, then the algorithm would've needed to be adjusted.

In the originally CUDA implementation, thread divergence and shared memory was not considered. Therefore, all the data that was used were all in global memory. Therefore, there was continuous access to global memory by all threads. The implementation was updated by taking in to consideration the warp size, which is 32 threads. That is, there was no thread divergence and all threads in each warp performed together with no conditions that would lead one thread to continue on a different path than another thread in the same warp. The implementation was updated once more by taking advantage of the shared memory of each synchronizing multiprocessor. In this case, the max of each thread block was saved in shared memory and ultimately sent to an array in global memory. This greatly decreased the access to global memory. The results of the execution can be seen below.

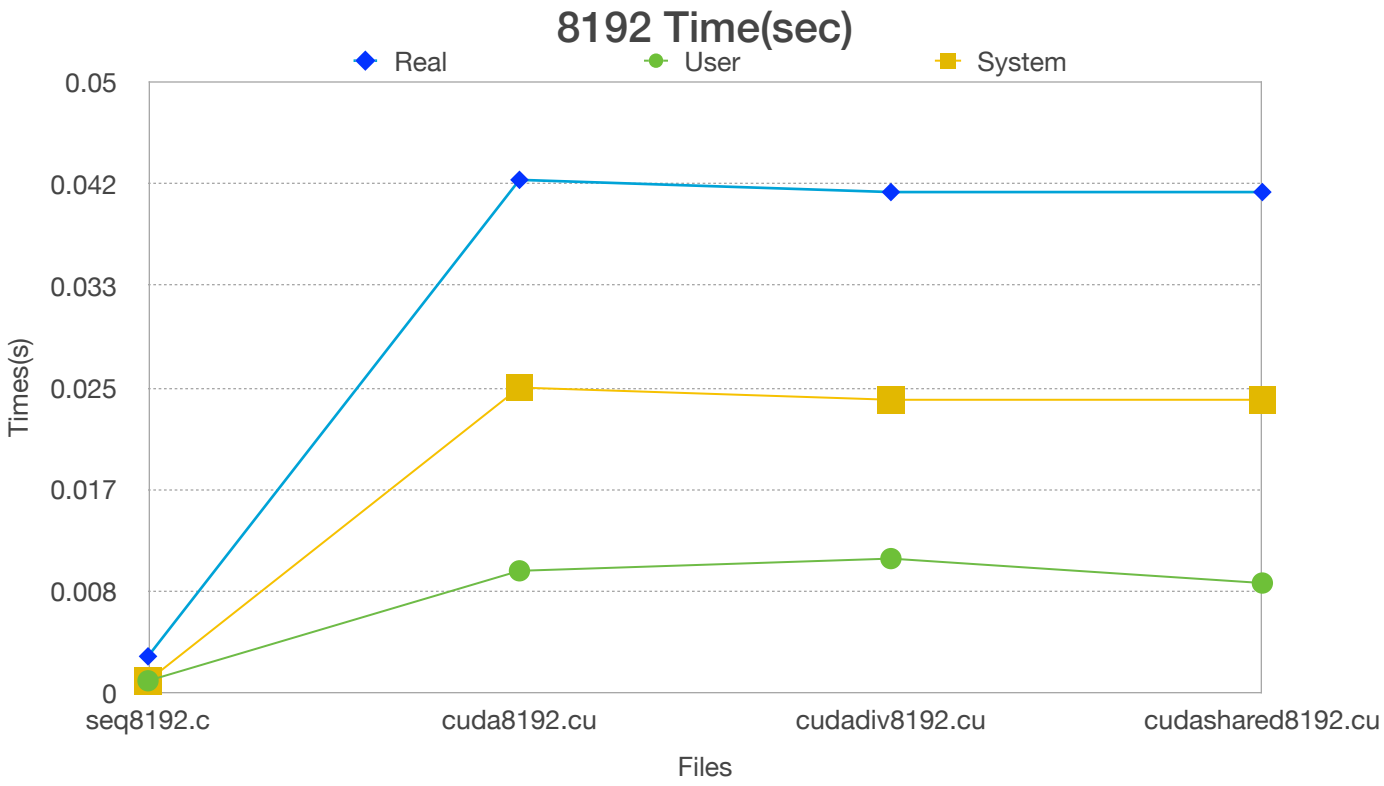
Data

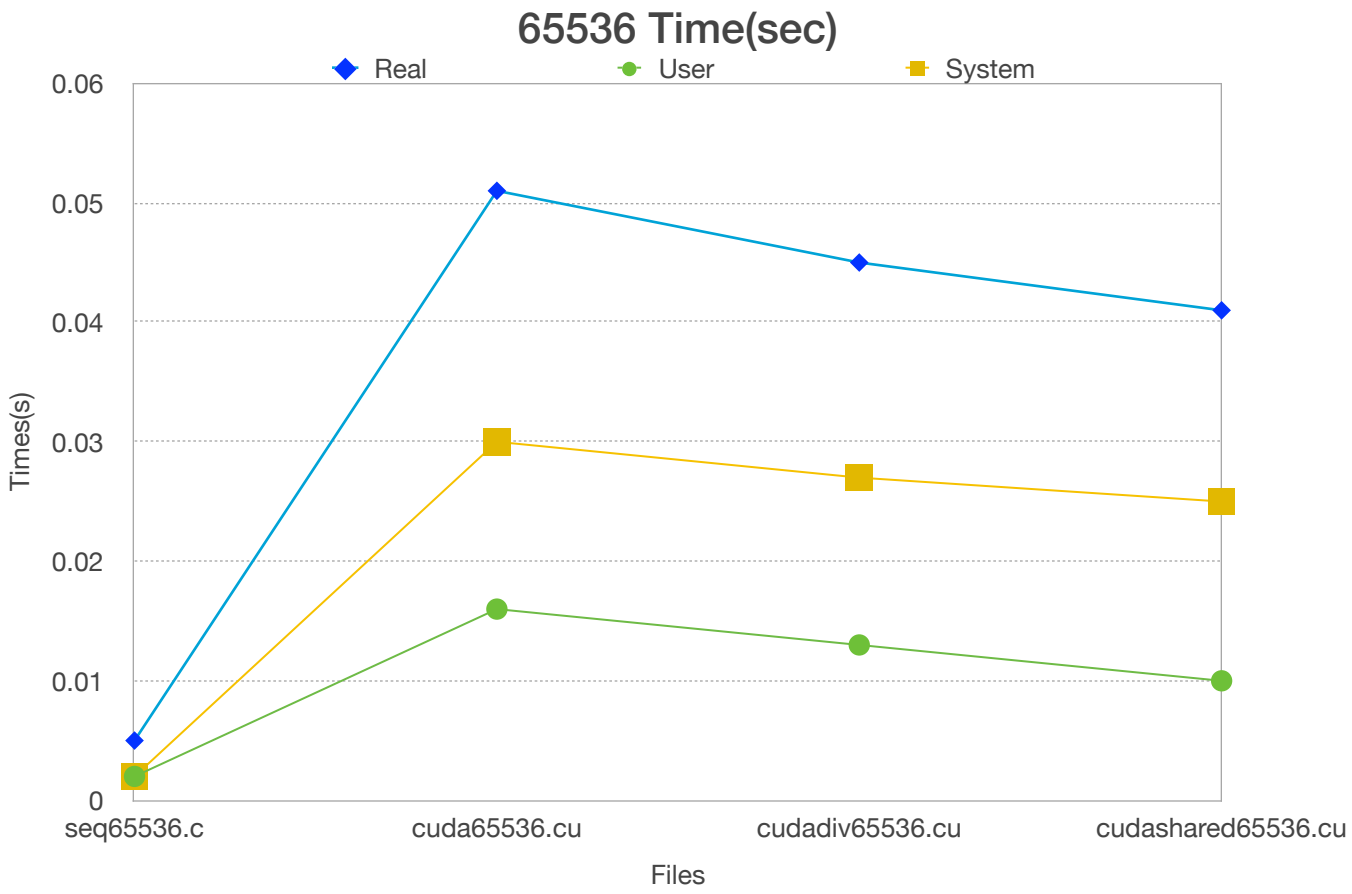
8192 Time(sec)

# of cities	Real	User	System
seq8192.c	0.003	0.001	0.001
cuda8192.cu	0.042	0.010	0.025
cudadiv8192.cu	0.041	0.011	0.024
cudashared8192.cu	0.041	0.009	0.024

65536 Time(sec)

# of cities	Real	User	System
seq65536.c	0.005	0.002	0.002
cuda65536.cu	0.051	0.016	0.030
cudadiv65536.cu	0.045	0.013	0.027
cudashared65536.cu	0.041	0.010	0.025





Conclusion

Since an array of 8192 or 65536 elements is really low, the sequential version of the implementation greatly outperformed that of its CUDA counterparts, no matter how much the CUDA version was modified(considering shared, thread-divergence). However, in both graphs, it is seen that as the CUDA versions updated to take advantage to shared memory and take thread divergence in to consideration, the better the implementation performed. This is not that obvious when the array size is 8192 but it is slightly more obvious when the array size was 65536. In any case, solving this problem would've have been more efficient if we used the sequential implementation. But it is obvious that as the number of elements increase, the greater the CUDA versions perform.