Kaylo Littlejohn - GPU Lab 2 Results

GPU Compilation Times:

Vector Size	Block Size	CPU Time	GPU Time (basic)	GPU Time (improved)
100	32	0	4.59	0
100	128	0	4.59	3.78
100	512	0	4.56	3.77
100	1024	0	3.77	4.32
100000	32	0	7.87	0
100000	128	0	6.34	0
100000	512	0	3.59	0
100000	1024	0	3.63	0
100000000	32	233.66	549.79	225.28
100000000	128	225.07	538.17	232.9
100000000	512	227.11	540.33	225.21
100000000	1024	225	538.97	224.37

Questions:

We can see that the code has a faster execution time when using shared memory. As the block size increases for a given vector size, the GPU utilizes more threads and the performance improves. When we try running the program with a vector size of 1,000,000,000, we killed the program because it took a very long time.

CudaMemset fills a defined amount of bytes in memory pointed to by a pointer with a constant integer value. We always set the block size as defined because we want to be able to directly use the blocksize in calculations in the kernel function . We copy the a array passed into the kernel into the shared memory because that array depends on previous and future positions, thus using shared memory prevents miscalculations based on values that have not yet been calculated. If the block size is different the vector size, the program should still work, although excess threads could be executed if the vector size is not a multiple of the block size. We use can choose to use constant memory for any data that does not change during the kernel's execution, which in this case is an integer value called constant. We use syncthreads to sync the progress of each thread after transferring the arrays to shared memory. We can improve our code further by using CUDA-memcheck to check if memory is being illegally accessed, which is possible at TID = 0 or TID = vector_size-1. This was exactly the case in our code.

For the Nvprof code, we can see statistics for each type of CUDA library call used in the program, such as min, max, calls, and average. We can speed up performance by trying to

reduce the average time each call takes to execute. In our created nvprof.log file, the metrics regarding throughput and hit rate could be of interest. Also, we noticed that the gld_efficiency and l2_l1_read_hit_rate metrics are both 66%. This means that 66% of the available global memory is being utilized. Gst_effeciency shows that 100% of the data stored into memory is in fact being stored. We tried to improve the code using shared memory for the a array, however in this case, the metrics did not change much and the efficiency did not improve. Shared memory is not really useful in this case