Jensen Eicher
Parallel Computing
4/9/20

HW 3 Write Up:

count.cu

After reading the assignment details and understanding the problem I began implementing count.cu. First, I wanted to design the Kernel. Following the slides, I setup a thread block size of 128 and created a global_index using the block dimensions, block id, and thread id (with respect to x (1D)). I chose to implement the count kernel using a shared variable called blockSum. I then used thread 0 to initialize blockSum to 0 and used a syncthreads to eliminate the possibility of a thread racing to change blockSum before its initialized. Next, I looped through the flat input array using the global index and when a 1 was found I atomically added 1 to that threads blockSum. Last, I used one last syncthreads to make sure each thread has finished updating their blockSum and then atomically added each blockSum to the output. In short, I used a shared variable to further increase the parallelization by allowing each threadblock to calculate its own blockSum (taking advantage of coalesced memory) and then atomically adding each blockSum to accumulate the total sum. I used cudaMallocs to allocate memory for the input and output variables, performed the data transfer from the host to the device, then created the block and grid variables that were used to launch the kernel. Lastly, I copied the data back to the host from the device.

transpose.cu

In the transpose problem I also started with the kernel. In this example we cant simply use a flat array since we care about the order of the matrix. This means we need two global indexes one for x and one for y (same as I and j (2D)). I then loop through the input (matrix) using both global indexes and perform the transpose. We could take advantage of shared memory to allow both the reads and writes from the thread blocks to be coalesced memory accesses. This could be achieved by reading and writing to global memory in a coalesced manner but using the shared memory to perform the transpose and the single non-coalesced memory access step. Since the assignment only requires one of the two programs to use shared memory, I decided to implement it in count.cu instead. The variable declarations and data transfers are similar to count.cu except we are now receiving a matrix as the output rather than just an integer. Launching the kernel has changed to a block of dimension 16 by 16 and the grid being a multiple of that, creating a square. This allows for the transpose to work well in the GPUs cache since the thread blocks are squares.