George McGurkin
EE155 / CS122: Parallel Computing
Lab 5: Computing CNN with CUDA

My implementation's timing results:

	4x4 filter	**8x8 filter**	**12x12 filter**
4Kx4K input	0.033 sec	0.035 sec	0.039 sec
8Kx8K input	0.133 sec	0.135 sec	0.139 sec

Questions:

- For a given filter size, did the 8Kx8K matrix take 4x longer than the 4Kx4K matrix as expected?

For each given filter size, we can observe that the total runtime was approximately 4x longer for a 8Kx8K matrix than it was for a 4Kx4K matrix. This makes sense, as the size of the input matrix dominates the size of the filter matrix, and as a result the time spent moving the data around is roughly quadrupled between 4K and 8K, as the 8K matrix contains 4x the number of elements as the 4K matrix. In addition, the increased size will also result in roughly 4x the number of thread blocks, increasing the computation time by 4x.

- For a given matrix size, did the 8x8 filter take 4x longer than the 4x4 filter, and did the 12x12 take 9x longer than 4x4 as expected?

Increasing the size of the filter did not result in a 4x time increase between 4x4 and 8x8 and a 9x increase between 4x4 and 12x12. While the filter size increased, the input matrix remained the same, and because the size of the input matrix dominates the size of the filter matrix, the time spent moving data remained roughly the same. The computation time did increase, as each thread performing the multiply accumulation for each output element was responsible for more multiplies. But because the computation time is dwarfed by the time spent moving the data, we only see a slight increase in the total runtime between the different filter sizes.

- If they did not, then what does this say about how long your program spent in various data-movement times vs. computation times? Is this consistent with what your more-detailed timing results told you?

As stated above, the timing data suggests that the bulk of the total runtime was spent moving the data as opposed to performing the actual computation, as we see a roughly 4x increase between the 4Kx4K and the 8Kx8K input matrix sizes, but no 4x increase between the 4x4 and 8x8 filter

sizes. Examining the more-detailed timing results supports this. For instance, one run of CNN using the 8Kx8K input matrix and the 12x12 filter spent 0.067s copying the data to the device, 0.005s computing on the GPU, and 0.064s copying the data to the host. Because the bulk of the time is spent moving data, the increased filter size doesn't impact the time as much, as it marginally changes how much data needs to be moved.

Design Choices:

My implementation works by breaking the input data into 32x32 chunks, and performing all the multiply accumulations possible that make full use of that block. Given a 32x32 block, and a filter size x filter size filter, only 32 - filter size + 1 x 32 - filter size + 1 output values can be calculated, as any threads with a thread index x or y greater than or equal to 32 - filter size + 1 would need to access data found outside of the shared memory. As a result, the approach was to use 32x32 thread blocks, where each thread loaded in a value of the input matrix, and only threads with x and y indexes less than 32 - filter size + 1 would perform the multiply accumulation. While this made it possible to neatly perform the computations that only make use of the data within the 32x32 block, the values not contained within the 32 - filter size + 1 x 32 - filter size + 1 block at the upper left corner of the input matrix would need to be loaded into another thread block's shared memory for their own computation, and as a result some values are loaded from DRAM twice (we know it is not more than twice because the largest filter size is 12, which is less than half of 32). However, this simplified the math, and eliminated the need for a critical section, so it was a cost I was willing to pay.

The checks within each thread determine what tasks each thread performs, in order to avoid duplicate computations and out of bounds accesses. First we check if the given thread indices maps to a value outside the input matrix. If not, that thread loads its corresponding input value from DRAM into shared memory. Next, we check if the thread's index is in the top filter size x filter size square in the top left corner of the matrix. If so, that thread loads its corresponding value of the filter matrix into shared memory. This way each value of the filter matrix is loaded by one thread in the thread block. Next to determine whether a thread is going to compute an output value, we first check if it is contained within the 32 - filter size + 1 x 32 - filter size + 1 square in the top left corner of the input block, meaning the filter computation relies only on values within the current memory block. Next we perform another check to see if the thread's index is contained within the output matrix. This handles the edge cases where we are dealing with thread blocks that are found on the very edge of the input matrix, enforcing that no out of bounds accesses occur.

We enforce that no memory-coalescing issues occur by assigning rI values using threadIdx.y and cI values using threadIdx.x. Because we access values in DRAM indexing via rI for rows and cI for columns, threads within the same warp store values in shared memory that are adjacent in

DRAM (as threads in warps share threadIdx.y, but differ with threadIdx.x). This applies for both storing the input matrix block and the filter matrix. We also avoid bank conflict issues. Our traversal of the filter matrix occurs in row major order. For each shared memory filter access, every thread in the warp accesses the same memory address, so no stalling occurs. For the input matrix block accesses, each thread of the same warp accesses a value from a different bank, as each access is offset by the thread's threadIdx.x value, which is different for each thread within a warp. As a result, no stalling occurs.