Parallelization Strategies

I eliminated all of the i, h, and w loops by having a 3-dimensional grid of threads.

Bias

The bias “loop” has nothing fancy; each thread just gets the one bias value it needs.

Convolution

The input array is cached into shared memory for each fixed value of i and j. This is because each input value is accessed multiple times by different threads (values of h, w) at different times (values of p, q). There is enough space on the GPU for this because the number of values scales linearly with the number of threads. I could have also cached the weights array but I was happy with my performance already. The threads each keep a private copy of the c values that they are calculating to avoid expensive memory accesses.

Max pooling & ReLU

To optimize the max pooling step, I make each thread calculate a 2x2 output of the convolution so that it can max pool right away without communicating with other threads. The ReLU loop is merged with the max pooling one but there aren't any parallelization strategies related to it.

Optimization

The most trivial optimization was to port the sequential code to the GPU. The threads communicate their values for c in a shared array in order to do the max pooling step, but I was able to eliminate that by having each thread calculate its 2x2 neighbors (which also cuts the number of threads by a factor of 4). This also meant that I had few enough threads that I could have each one compute only a single i,h,w value of the output. At this point I merged the ReLU and max pooling loops, and swapped the order they were applied (they commute) to get rid of 3 max operations. This wasn’t quite enough for an A, so I made a shared array to cache the values of the input array (for a particular j). This reduces memory accesses because the threads in a block access a value in the input multiple times as they vary p and q.

Dimensions

I used 2x8x16=256 threads per block and 128x14x7=12544 blocks per grid. This checks out with the specs for the Tesla M60 GPUs, which can have up to 1024 threads per block. Each SM can have 2048 threads, which is 8 blocks (within the max of 32 per SM). There are 2048 CUDA cores to use, and my blocks can only saturate 12544/8=1568 of them so there are not too many blocks.