Lab 3 Report

Performance

Approach	GFlops	Time (s)
`cnn_seq`	13.3	12.4
`cnn_gpu`	729	0.226

General Strategy

I chose to have each thread be in charge of one final output element `output(i,h,w)`. Since the final step of the algorithm is to perform a maxpool among a 2×2 grid, each thread was in charge of producing 4 intermediate outputs (`c00`, `c01`, `c10`, `c11`) from the convolution and ReLU steps.

I used a 3D grid with 3D blocks to determine each thread's corresponding output element:

```
int i = blockIdx.z * blockDim.z + threadIdx.z; // output channel
int h = blockIdx.y * blockDim.y + threadIdx.y;
int w = blockIdx.x * blockDim.x + threadIdx.x;
```

In order to determine the region of the input that affects the output element, each thread multiplies `h` and `w` by 2 (to reverse the effect of the maxpool downsampling).

For the convolution step, each thread initializes its own `c00`, `c01`, `c10`, `c11` floats to `bias[i]`. Then, it performs the convolution with 3 nested for loops that iterate through each dimension of the kernel (starting with `kNum` iterations in the outer loop and then `kKernel` iterations for each of the two inner loops) and applies the convolution (by multiplying the appropriate weight with the corresponding region of the input). For each iteration of the innermost loop, all four intermediate outputs `c00`, `c01`, `c10`, `c11` are updated appropriately.

Finally, each thread performs ReLU on each of `c00`, `c01`, `c10`, `c11` and sets `output(i,h,w)` to the maximum of these 4 values.

I chose this general strategy because it eliminated the need for inter-thread communication and/or dependencies. When I first approached the problem, I chose to have each thread be responsible for one output element from the convolution step, but then quickly realized that performing the maxpool operation would require complex synchronization among threads. With my current approach, each thread can operate independently.

Optimizations

I applied some important optimizations in my convolution step below:

```
int h_base = h * 2;
int w_base = w * 2;

float c00 = bias[i], c01 = bias[i], c10 = bias[i], c11 = bias[i];

for (int j = 0; j < kNum; ++j) {
   for (int p = 0; p < kKernel; ++p) {
      for (int q = 0; q < kKernel; ++q) {
         int ih0 = h_base + p;
         int ih1 = h_base + 1 + p;
         int iw0 = w_base + q;
         int iw1 = w_base + 1 + q;

         float w_val = weight(i,j,p,q);

         c00 += w_val * input(j,ih0,iw0);
         c01 += w_val * input(j,ih0,iw1);
```

```
                c10 += w_val * input(j,ih1,iw0);
                c11 += w_val * input(j,ih1,iw1);
            }
        }
    }
```

First, I explicitly used the float variable `w_val` so that the compiler would be more likely to allocate the value in a register (from global memory). Most importantly, though, I applied a loop transformation (fusion) so that each convolution output could be accumulated in each iteration of the innermost loop. This allowed `w_val` to be reused from the register for better memory performance. It also reduced the control overhead as each iteration of the loop resulted in more "work" being done. Finally, I used the above loop order so that there might be better cache performance when accessing the `weight` matrix across iterations of the inner two loops.

Dimensions Discussion

As I have discussed previously, each thread is in charge of one output element `output(i,h,w)`, and I chose to use the three block and grid dimensions for determining these values.

First, I decided to use the `z` dimension for determining a threads output channel `i` (of which there are 256 total), so I knew that `blockDim.z` * `threadDim.z` should equal 256. I then decided to use the `y` dimension for determining `h` so I knew that `blockDim.y` * `threadDim.y` should equal 112. The same logic applied for using the `x` dimension to determining `w`.

The T4 GPU can support up to 1024 threads per block. It also has 40 Streaming Multiprocessors, each with 64 cores. I played around with different dimension combinations that had less than 1024 threads per block and met the constraints outlined above. I found good performance with the following dimensions:

	X	Y	Z
Block	8	8	4
Grid	14	14	64

In this case, there are 256 threads per block (which is supported), and 12,544 total blocks. Having 256 threads per block means there can be 8 warps from a single block without any that are partially full. Furthermore, because my block size is a multiple of 2, warps can be scheduled to maximize all 64 CUDA cores in an SM every clock cycle. While my configuration implies 12,544 blocks (which is more than the total supported by all 40 SM's at once), it is not a huge problem since the GPU will just schedule a block to an SM once one finishes. Thus, the GPU specification does corroborate my chosen dimensions.