# Computer Science 133, Lab 3

Michael Wu
UID: 404751542

March 1st, 2019

## 1 Parallelization Strategy

In order to parallelize this code, I began by working backwards from each element of the output array. I set the dimensions of the global work group to be equal to the dimensions of the output array in order to allow for the fastest kernel possible. From the max pooling step, I saw that an output element was simply the max of four elements in the convolution array. I included the ReLU step by taking the max with zero after the max pooling in order to reduce the total amount of comparisons. Then for each work item I calculated 4 elements of the convolution array and applied this max function. Performing convolution was the most computationally costly step of the kernel, as it required three nested loops. These four convolution elements are never reused in any other output element, so I could store the intermediate convolution calculations on the stack. When I tested this approach I obtained a speed of about 100 GFlops.

## 2 Optimizations

I further optimized my code by performing loop unrolling. I knew that each processor could do up to 16 operations in parallel, but I was only calculating 4 elements of the convolution array at a time. So I decided to calculate 4 elements of the output array at a time which allowed me to calculate 16 elements of the convolution array in parallel. I chose to unroll along the innermost index in order to gain the best cache efficiency. I found this to be the most optimal amount of unrolling. Unrolling 8 times reduced

performance by increasing the size of each work item, while unrolling 2 times did not offer enough parallelism. In the end I obtained upwards of 200 GFlops with 4 times unrolling.

# 3 Performance Comparison

My code obtained a performance of 211.68 GFlops with an execution time of 0.776722 seconds. The sequential code obtained a performance of 11.5451 GFlops with an execution time of 14.2412 seconds. This meant that my code had a speedup of 18.33 times over the sequential version. My code has $256 \times 112 \times 28$ work items, since I created a work item for every four elements in the output array. I found that setting my global work group dimensions to $(256, 112, 28)$ and my local work group dimensions to $(2, 1, 1)$ gave me the best performance. These local work group dimensions allow for the best cache performance for this problem. If I set the local work group to size $(1, 1, 1)$ then memory is not used efficiently and I obtain a performance of 31.3487 GFlops. Increasing the local work group dimensions any higher does not lead to better performance.

I wrote an additional kernel that split the problem along the outermost dimension of size 256 in order to test my performance on a varying number of work items. My global work group size was one dimensional and equal to the number of work items. My local work group size was 1. The result of my testing is shown below.

| Work Items | Speed (GFlops) |
|---|---|
| 1 | 8.3467 |
| 2 | 10.2398 |
| 4 | 21.2904 |
| 8 | 44.5728 |
| 16 | 44.5946 |
| 32 | 44.5989 |

The performance increases fairly linearly until after 8 work items where it stagnates. In order to allow varying work items I had to make my code less efficient so I was not able to match my most optimized speeds.

# 4 Challenges

Challenges I faced included figuring out the array indices and memory management. When trying to set up a minimum working example, I had to do a lot of pointer arithmetic since the inputs to the kernel were multiple dimension arrays that were collapsed down to one dimension. I also could not fit the entire convolution array into memory, which prevented me from running code that was similar to the sequential code. I overcame this by writing out the kernel for a single output element. I also tried manually splitting the output into different partitions and wrote a lot of inefficient code that way. I found that it was best to simply have a high number of work items.