Graham Wing

UID: 004738350

# Lab 4 Report

## Parallelization Strategy:

My parallelization strategy was largely identical to the approach I took in Lab 3. The parallelization approach for initializing values and computing the respective convolutions was identical to the previous lab. Each work item is assigned a unique set of outputs to compute. Work items are then executed in parallel as defined by the global work size and synchronized with each other as defined by the local work group size. The same approach can be said for parallelizing the max poolings. However, while the broad approach to parallelization is the same, there some slight differences. The primary difference is the approach to memory management. Because GPU architectures allows for better sharing of memory between cores than CPU architectures, I was able to unroll more than the previous lab. Computing 16 (single layer, 2 adjacent rows, 8 adjacent columns) convolutions per work item compare to the 8 convolutions of the previous lab.

The biggest difference was active memory management within a local group. Because I fixed the local work group size, I was able to parallelize movement of the input array (the largest of the arrays in use). For every layer of the input used, I had 256 concurrent threads pull in the 1280 required values per input layer needed by each work group. This drastically improved parallelism over the CPU since CPUs have a much smaller degree of concurrency.

## Optimizations:

- Identical to the previous lab, I first allocated a work item per convolution. This allowed better utilization of the GPUs resources (given that it has 2048 cores) and provided basic parallelization, but poor memory management.
- I then assigned more convolutions per work item. Using the same code as the previous lab (single layer, 2 adjacent rows, 4 adjacent columns) I was able to achieve 350 GFLOPs (immediately better than the prior lab). This is entirely due to better use of memory. Since adjacent convolutions share values, each time a value was loaded into a core's memory, it was more likely to be used multiple times during the execution of a work item. By increasing the number of assigned items (single layer, 2 adjacent rows, 8 adjacent columns), I increased the performance to 480 GFLOPs.
- I next explicitly loaded sections of input into local memory for use within a local work group. My initial approach was to load as much as possible into memory (64 by 8 by 20). This effectively blocked my loops. Since each computed 16 convolutions (160 values shared between 16 convolutions), every 64 loops I had to load more values in. Fortunately, I could parallelize this between local work items based on local dimension 0. Unfortunately, this only improved my performance to 510 GFLOPs.
- By playing with the values in params, I found that I could incorrectly achieve 1200 GFLOPS by setting the local dimensions to 32 by 8 by 1. Since I blocked the loop in dimension 0, I could not correctly use these params. This led me to change my approach for loading memory. I decided to trade full utilization of local memory for increased dimension utilization. By loading values each step through the layer loop (8 layers per local workgroup, 160 values per layer, 32 threads per layer, 5 values per thread) in parallel I was able to improve my performance to 1440 GFLOPs. A massive improvement over the previous approach.

**Work Group Size:**

The best performance I achieved was with 784 work groups and 256 work items per work group ((256 by 56 by 14) / (32 by 8 by 1)). Looking at our class notes, the Tesla M60 has 16 multiprocessors and 128 CUDA cores per multiprocessor. The numbers do not identically match since the work size is rather large, but the resulting performance is intuitive based on these values. Since 256 work items are assigned to each work group, each multiprocessor must execute 2 kernels per work group. Each multiprocessor must execute 49 work groups. Since the number of work groups and work items per group are multiples of the multiprocessors and cores per multiprocessor, no cores are idling during execution. This allows for better performance. Therefore, the numbers match each other.