Jason Lai
204995126
DIS 1A

CS133 Lab 3 Report

Parallelization Strategies:

Throughout the course of the lab, I tried numerous different parallelization strategies, many of which ended up being too difficult for me to improve or not helpful at all (see Challenges section at the end for details). In the end, I went with the strategy that I did because I felt like it was a relatively natural approach to parallelizing this convnet algorithm. In my final implementation, each step of the convnet process: setting the bias, convolution, ReLU, and max pooling is placed in its own nested for loop with indices hh and ww. These nested for loops pertain to the size of a "subtile" that is essentially a piece of the intermediate feature map that a given kernel will compute. In order to parallelize this, I essentially created as many work items as there are subtiles and assigned each one a subtile to work on. Then each subtile goes through the steps of the convnet process. More concretely, I divided the 224 x 224 feature map into 16 x 32 tiles, and I assigned each tile to a work group for a total of 98 work groups. Then each 16 x 32 tile is broken down further into 4 x 32 subtiles that are assigned to a work item in a given work group; this means there are 4 work items per work group, which process their subtiles in parallel for every step: setting the bias, convolution, ReLU, and max pooling.

Optimizations:

Loop Fusion / Tiling:

The first strategy that I employ fusing the i loops and tiling the h and w loops, although this is mostly to keep from blowing the stack and running out of resources. However, in my final implementation I still maintain tiling wherein each work group is assigned a tile of size 16 x 32. These tiles are further broken down into subtiles of size 4 x 32, to assign to each of the 4 work items in the work groups for execution in parallel.

Loop Permutation:

Next I used loop permutation on the convolution loop where the 5 x 5 filters are applied to the input image. The loop permutation I performed was to move the j, p, and q loops outside of the hh and ww loops, then swap the p and q loops. The reason for this was because we see that the weight array is mostly iterated on j, so the same element of weight will be used multiple times, but we access it multiple times as well. We use loop permutation so that we can reuse the weight filter for a given j by only accessing it once in the outer loops.

Manual Vectorization:

To further optimize the convolution loop, I used manual vectorization to load rows of the filter array and input image into vector registers of size 16 (float16). These vectors are then multiplied

with each other and stored appropriately in their respective intermediate arrays. This optimization has a similar effect to loop tiling.

Loop Unrolling:
I then “unrolled” the i loop so that I could run the same subtile through the convnet for different channels. To do this, I initialized 8 different arrays of the same size and then unrolled the i loop by 8 iterations, where each array is assigned to an iteration (channel) to hold to values of. This means for a given subtile, we compute its result for 8 channels at a time.

OpenCL Parallelization:
To parallelize my program, I first decided what size tile I wanted to use, which ended up being 16 x 32 (what could fit into the shared memory of work groups that also divides evenly in the intermediate feature map and output image). Each of these tiles is assigned to one of 98 work groups, as there are 98 16x32 tiles in the 224 x 224 feature map. These tiles are then further divided into subtiles of size 4 x 32 for each of the 4 work items in each work group to compute.

Performance Comparison and Scalability:

For performance measurements and scalability, my program’s implementation scales according to how fine the granularity each 16 x 32 tile is broken down into subtiles. In other words, if we want to increase the number of work items in a work group, we must break the 16 x 32 tile each work group is assigned into more pieces in order to give the additional subtiles meaningful work to do. For instance, if I currently have 4 work items in each work group with a subtile size of 4 x 32, but I want to double the number of work items to 8, then I must decrease the subtile size to 4 x 16, so that the 16 x 32 tile is broken into 8 pieces; one for each work item. This is how I adjust to measure my program’s performance in regards to the number of work items.

Version / Trial	1	2	3
Sequential	14.2738 s 11.5188 GFlops	14.2869 s 11.5083 GFlops	14.2691 s 11.5225 GFlops
Subtile: 4 x 32 Global: ‘56 7’ Local: ‘4 1’ 98 groups, 4 items each	0.868481 s 189.315 GFlops	0.868096 s 189.399 GFlops	0.868619 s 189.285 GFlops
Subtile: 4 x 16 Global: ‘56 14’ Local: ‘4 2’ 98 groups, 8 items each	0.90647 s 181.381 GFlops	0.907966 s 181.082 GFlops	0.914513 s 179.786 GFlops
Subtile: 4 x 8 Global: ‘56 28’ Local: ‘4 4’	1.79938 s 91.3743 GFlops	1.81787 s 90.4445 GFlops	1.89362 s 86.8267 GFlops

98 groups, 16 items each			
Subtile: 4 x 4 Global: ‘56 56’ Local: ‘4 8’ 98 groups, 32 items each	3.64899 s 45.0581 GFlops	3.63626 s 45.2159 GFlops	3.67188 s 44.7772 GFlops
Subtile: 4 x 2 Global: ‘56 112’ Local: ‘4 16’ 98 groups, 64 items each	7.3416 s 22.3952 GFlops	7.3092 s 22.4945 GFlops	7.22553 s 22.755 GFlops
Subtile: 2 x 2 Global: ‘112 112’ Local: ‘8 16’ 98 groups, 128 items each	7.40242 s 22.2112 GFlops	7.46604 s 22.0219 GFlops	7.43644 s 22.1096 GFlops

Note that these numbers are performance measurements taken on the m5.2xlarge instance of AWS. I observed after running my program on Gradescope that my program runs considerably better on the c5.2xlarge instance of Gradescope.

The first thing to note is that the global/local work size that gives me the best performance is ‘56 7’ and ‘4 1’ respectively, which means that there are 98 work groups with 4 work items each. Compared to the sequential version of the program, we see that my parallelized implementation provides a drastic increase in performance, as we have that the average number of floating point operations per second of the parallel version across the 3 trials is 189.333 / 11.5165333 ≈ 16.44 times more than the average number of floating point operations per second for the sequential version.

As for the scalability, we see that as we increase the number of work items per work group, while maintaining the same number of work groups, we have that the performance scales relatively linearly in the range from 8 to 64 work items in each work group, wearing the increased number of work items actually worsens performance. Beyond this range, we see that both the performance increase or decrease ceases to scale linearly and instead plateaus. Due to the way I programmed this lab, I surmise that the reason for this odd scaling behavior is due to resource limitations, as resources are likely to already be fully utilized at my lowest work item count, so increases to the amount of work items would thereby increase overhead, so we have performance dropoff as work items scale up.

Challenges:

This lab proved to be very challenging. I tried numerous different implementations, but found that each was too difficult for me to improve, and thus I would opt to try other avenues of approach. Initially, I tried to improve the performance of the sequential program as much as possible before I attempted to parallelize. This entailed applying manual vectorization and loop permutation to the convolution loop and unrolling the i loop for channels. Furthermore, I wanted

to combine the bias, ReLU and max pooling steps since there is no dependency between the bias and convolution steps, so we could feasibly aggregate these steps into a single loop. However, in order to do this, we must zero-initialize the buffer in which we store the intermediate feature map, but my implementation uses a __local scoped array, which can't be zero-initialized. However, we can zero-initialize __private scoped arrays, but upon changing my arrays to __private, I found that this simple switch from __local to __private hurt my performance. My only guess as to why is because the large arrays that were stored in the shared memory of the work groups are too big for the local memories of the individual work items. I also tried many different ways of using work groups/work items to achieve better parallelism, but the lack of knowledge on how to use OpenCL and lack of documentation for some of the bugs I was encountering made this very difficult to work with and understand.