Jason Lai
204995126
DIS 1A

CS133 Lab 4 Report

<u>**Performance Summary:**</u>

Version / Trial	1	2	3
Sequential	15.9017 s 10.3396 GFlops	15.8858 s 10.3499 GFlops	15.8639 s 10.3642 GFlops
Parallel Global: ‘64 112 16’ Local: ‘4 8 2’ 1792 groups, 64 items each	0.073032 s 2251.3 GFlops	0.073091 s 2249.48 GFlops	0.073135 s 2248.13 GFlops
Speedup	15.8838 / 0.073086 ≈ 217.33x speedup		
Performance Improvement	2249.63666667 / 10.3512333333 ≈ 217.33x more Flops		

From the table above, we see that the performance of my OpenCL implementation with an NVIDIA GPU (Tesla M60) is approximately 2250 GFlops with an execution time of about 0.073 seconds. Averaging the results of the trials for the sequential and parallel programs, we see that the speedup of the parallel OpenCL implementation is approximately 217.33x over the sequential implementation. As for the performance, we also see that the parallel implementation performs approximately 217.33x more floating point operations per second than the sequential implementation.

Work Group and Work Items Per Group Table (*Bonus*):

The table below shows the performance for different numbers of work groups and work items per work group for 3 × 3 = 9 different configurations including the one that provides the best performance (1792 work groups and 64 work items per work group).

Work Groups / Work Items Per Group	32	64	128
896	4.42765 s 37.1341 GFlops	0.355574 s 462.398 GFlops	0.074137 s 2217.74 GFlops
1792	0.357569 s 459.818 GFlops	0.073258 s 2244.35 GFlops	0.100015 s 1643.92 GFlops
3584	0.084161 s 1953.6 GFlops	0.100426 s 1637.19 GFlops	0.144966 s 1134.17 GFlops

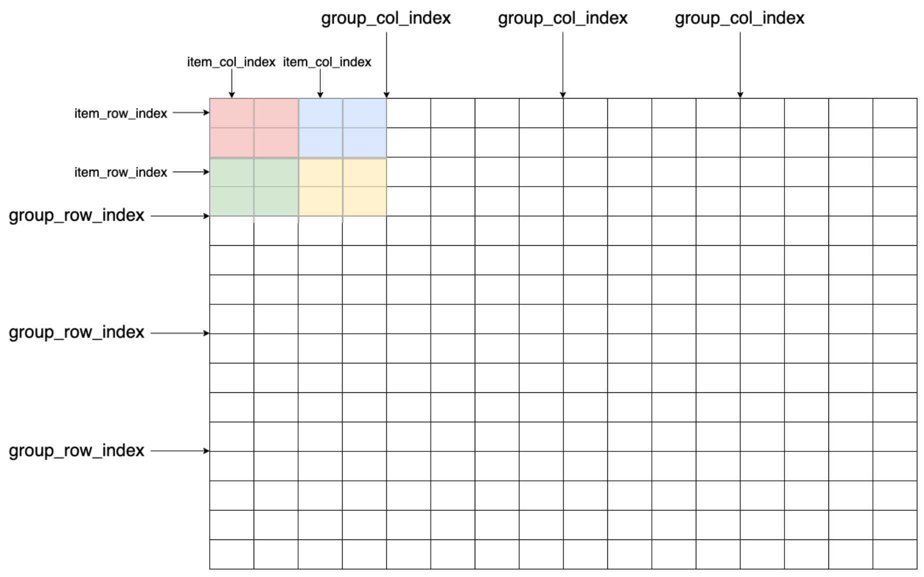
To change the number of work groups and work items per work group, I adjusted just the first dimension of the `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` parameters only. This is effectively changing the number of channels that each work group and each work item handles, since the first dimension is purely for splitting the `i` loop (used for channels) in my implementation. I did this because there are many different ways to achieve the same configuration for a number of work groups and work items per work group when adjusting any of the three dimensions, so I wanted to restrict the configuration space by limiting changes to a single dimension and the first dimension “plays the nicest with the numbers”.

From the table above and our performance summary table, we see that the best performing configuration uses the parameters `'64 112 16'` and `'4 8 2'` for `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` respectively which amounts to 1792 work groups and 64 work items per work group. Looking at the various configurations that surround it in this 3 x 3 solution space, we see that this best performing configuration appears to be a local maximum in regards to its performance versus its configuration.

Parallelization Strategies:

I assume that parallelization strategies are different from optimizations (e.g. loop unrolling, permutation, etc.), where “parallelization strategy” refers to how work is distributed and computations are done, which the lab spec implies can vary for the different loops in each kernel executed by a thread.

In this lab, the parallelization strategy I used is about the same as the one I used in lab 3, wherein there is no specific strategy that I used for the different loops (bias, convolution, ReLU, max pooling). Instead, the way that I parallelize the `cnn` program is by having each work group assigned a tile of the intermediate feature map to compute and each work item is assigned a “sub-tile”, which is essentially a piece of this larger tile that the work group that work item belongs to is assigned. These sub-tiles are then computed by each work item through every stage of the convnet process i.e. the bias, convolution, ReLU, and max pooling loops are all performed on this sub-tile and only this sub-tile in each work item. In my program, the dimensions of the tile each work group is assigned is defined by the macros `i_tile`, `h_tile`, and `w_tile`, where the tile has dimensions `i_tile` × `h_tile` × `w_tile`. The same principle follows for the sub-tiles of the work items, where the sub-tile dimensions are given by the macros `i_subtile`, `h_subtile`, and `w_subtile`, which are used to initialize a 3D `__private` array for each work item. This strategy is the same as the one I used in lab 3 and I, again, chose this strategy because it was what I felt was a very natural way of parallelizing the convnet problem.



The image above helps illustrate how this parallelization strategy works in the 2D case. In the image above, if we assume that we have a 16 × 16 intermediate feature map with only a single channel to compute, then we assign each work group a 4 × 4 tile to process. Then, if each work group has 4 work items, we have each work item handle a 2 × 2 sub-tile of the larger 4 × 4 tile. In practice, each work group is actually assigned a "prism" or 3D array since we have that each work group will also handle multiple channels for its tile.

The arrows labeled `group_xxx_index` in the image were variables I had initialized in an earlier implementation, but they essentially demarcate where the starting indices of each tile that each work group is assigned are. They can be found by multiplying a given group's ID for a dimension of the work group parameters by the appropriate tile dimension macro `x_tile` e.g. `group_row_index` would be computed by multiplying `get_group_id(1)` by `h_tile`. These group indices give us the offset for each work item's sub-tile starting index in a given dimension which are indicated by the arrows with the labels `item_xxx_index`. To compute these starting indices we find the sum of the appropriate `group_xxx_index` value and the product of a work item's ID for that dimension and the sub-tile dimension macro `x_subtile` i.e. `item_row_index` = `group_row_index` + (`get_local_id(1)` * `h_subtile`).

The reason I do it this way is because by dividing the `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` parameters into three dimensions, one for each dimension of the intermediate feature map, and computing the starting indices of a sub-tile for each dimension, this makes applying this parallelization strategy to each convnet stage's loop easier. This is because for the bias, convolution, ReLU, and max pooling loops, each loop has an `i`, `hh`, and `ww` loop, which represent each of the three dimensions: channels, rows, and columns of a work item's sub-tile respectively. Thus, we define the bounds of the `i`, `hh`, and `ww` loops of the bias, convolution, ReLU, and max pooling layers by defining the size of the sub-tiles with the `i_subtile`, `h_subtile`, and `w_subtile` macros and get the position in the intermediate feature map that a given sub-tile pertains to by computing the starting indices of that sub-tile in this manner. This makes applying the parallelization strategy to all of the loops easy, and also makes testing different configurations of parameters for work groups and work items easier with minimal changes to the code.

<u>Optimizations:</u>

OpenCL Parallelization / Loop Tiling:

This optimization was already described in the **Parallelization Strategies** section, but to reiterate, we parallelize the convnet program by assigning each work item a sub-tile of the intermediate feature map to work on. This optimization can be thought of as being composed of two parts: the loop tiling part and the parallelization part. The loop tiling part is deconstructing the intermediate feature map into tiles; this is done initially in the baseline sequential implementation by tiling the `h` and `w` loops with inner `hh` and `ww` loops with smaller bounds, so that each iteration of the outer `h` and `w` loops is a different tile of the feature map that is computed by the inner `hh` and `ww` loops. The parallelization part involves distributing these tiles to work items, so that instead of a single thread computing all tiles, multiple threads can compute independent tiles via said work items. This effectively removes the outer `h` and `w` loops entirely and also tiles the `i` loop as well, so that each work item is associated with its own specific tile of set dimensions.

Loop Unrolling:

This optimization uses the pragma `#pragma unroll` on the `i` loop of each layer of the convnet program: the bias, convolution, ReLU, and max pooling loops. The pragma unrolls the `i` loop so that for each element of a work item's sub-tile that is processed at each layer, that element is computed for the number of channels denoted by `i_subtile` starting from the channel index denoted by `item_channel_index`. Doing this can reduce the control overhead of the `for` loops, as described in discussion, and increases performance greatly possibly due to the sheer amount of iterations that the `i` loops amount to given the problem size and amount of work items executed in parallel.

Loop Permutation:

This optimization entails performing loop permutation on the convolution layer of the convnet program, which is notably what I found to be the greatest bottleneck in the algorithm. The most important permutation performed on the loops `i`, `hh`, `ww`, `j`, `p`, and `q` that comprise the convolution layer is moving the `j`, `p`, and `q` loops to the outermost of the level of the loops. This provides a massive improvement to performance, because it improves locality of global memory accesses to the `weight` and `input` arrays (declared as read-only global variables by `__constant`), which iterate on the values `j`, `p`, and `q`. When the `j`, `p`, and `q` loops were the innermost loops, this made performance poor because, to produce an element of the sub-tile for the intermediate feature map, a 5 × 5 filter is retrieved from the `weight` array for multiple different channels, which requires multiple fetches from global memory due to cache misses; the same follows for accesses to the `input` array. By using permutation to move the `j`, `p`, and `q` loops to the outermost layers, we can iterate less on these values and instead reuse the same resources from the `weight` and `input` arrays for the entirety of the sub-tile, thereby reducing the amount of global memory accesses, which are very slow compared to accesses to local memory which the sub-tile is contained in.

Parameter Adjustment:

Although not exactly a "formal" optimization, fine tuning the number of work groups and work items per work group through `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` can have significant impacts on performance. The combination that I found to have the best performance is `'64 112 16'` and `'4 8 2'` for `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` respectively, which amounts to 1792 work groups and 64 work items per work group. My best understanding of why this configuration gives me the best performance that I have found is because the Tesla M60 GPU we use has two chips, each with 16 streaming multi-processors (SMs) and 2048 CUDA cores. However, our AWS g3s.xlarge instance only has one chip to work with, so we're left with 16 SMs and 2048 CUDA cores divided amongst them (128 threads per SM). The 1792 work groups are assigned as thread blocks to each SM by the compute work distributor. The threads within a block are then divided into thread warps composed of 32 threads each i.e. the 64 work items are separated into 2 groups and executed concurrently. According to some online research, I found that SMs can execute multiple thread blocks at a time, so with 128 threads per SM, we have 4 warps per SM, so two thread blocks can be executed at the same time per SM.[1] This means that there are up to 32 of the 1792 work groups being computed concurrently (16 SMs, 2 groups per SM), and by extension, 2048 work items being computed concurrently (64 work items per each of the 32 groups). Therefore, 2048 sub-tiles of the intermediate feature map are being processed in parallel at a time, which gives me the best performance that I have found, although I suspect there are configurations of work groups and work items per work group that yield even better performance that I have not found.

Performance Evaluation (*Bonus*):

The table below details the performance comparisons for different optimizations that I applied. For each optimization, I used my best performing version of the program that includes the optimization (given by the "With" row for that optimization) as a baseline, and then compare this version with a version of the program without that optimization ("Without") to show the difference in performance.

Optimization	Version / Trial	1	2	3
`get_global_id()`	With	0.073175 s 2246.9 GFlops	0.073063 s 2250.34 GFlops	0.073266 s 2244.11 GFlops
	Without	0.073847 s 2226.45 GFlops	0.073939 s 2223.68 GFlops	0.073268 s 2244.05 GFlops
Loop Permutation	With	0.073197 s 2246.22 GFlops	0.073769 s 2228.8 GFlops	0.073634 s 2232.89 GFlops
	Without	3.54504 s 46.3794 GFlops	3.43104 s 47.9204 GFlops	3.52651 s 46.6231 GFlops
Loop Unrolling	With	0.073104 s 2249.08 GFlops	0.073677 s 2231.59 GFlops	0.07362 s 2233.32 GFlops
	Without	1.32408 s 124.175 GFlops	1.32531 s 124.059 GFlops	1.3253 s 124.06 GFlops
“Loop Fission”	With	0.07321 s 2245.82 GFlops	0.073326 s 2242.27 GFlops	0.073726 s 2230.1 GFlops
	Without	0.089768 s 1831.57 GFlops	0.089457 s 1837.94 GFlops	0.090284 s 1821.11 GFlops
Memory Alignment* **	With	0.089757 s 1831.8 GFlops	0.089605 s 1834.91 GFlops	0.089613 s 1834.74 GFlops
	Without	0.094423 s 1741.28 GFlops	0.094282 s 1743.88 GFlops	0.094519 s 1739.51 GFlops

***Note: there is a caveat to the memory alignment optimization (read below).

Starting from the top, one optimization I applied was using the `get_global_id()` API in lieu of the `get_group_id()` API to compute the starting indices of each work item’s sub-tile. The increase in GFlops appears to be rather small (only about 15 - 20 GFlops), and it was hard to gauge whether this was actually an improvement to performance or merely placebo. However, I believe I noticed higher numbers of GFlops appearing more frequently. This could be attributed to the `get_global_id()` API somehow being faster than the `get_group_id()` API, but in retrospect, it’s more likely that the increase in GFlops is due to the fact that I have more floating point operations to compute in the kernel. This is because I calculate the value given by the `get_group_id()` API by using the equation `get_group_id()` = (`get_global_id()` - `get_local_id()`) / `get_local_size()`. This operation could be nominal for execution time, thereby leading to more floating point operations executed per second.

For the loop permutation optimization, a more detailed explanation for why this increases performance is detailed in the **Optimizations** section, but to reiterate, we permute the convolution layer loop by moving the `j`, `p`, and `q` loops to the outermost level of nesting. This improves the locality of resources cached from global memory for the `weight` and `input` arrays, which iterate on the indices of `j`, `p`, and `q`. This is because, by having these loops on the outermost level, we reuse the same cached data of the `weight` and `input` arrays to compute the entire sub-tile of the feature map for a single channel (the `j` value), rather than computing each element of the sub-tile individually for all channels, which would require different elements of the `weight` and `input` arrays for different channels and result in many cache misses and therefore many slow global memory accesses. From the results, we see that this reduction in global memory accesses provides an improvement in the order of about 48x more floating point operations per second.

Next, the loop unrolling optimization is also explained in the **Optimizations** section, but again, loop unrolling uses the unroll pragma of OpenCL to unroll iterations of an immediately following loop. In this case, I use it to unroll the `i` loop for each layer of the convnet program, so that elements of the sub-tile of the feature map are calculated for more channels in each iteration of respective outer loops. This can effectively reduce the control overhead for `for` loops, and increase the overall performance. From the results, this optimization improves performance by about 18x more floating point operations per second.

The next optimization: “Loop Fission” is more of a significant code change that my program underwent as I was writing this section than an optimization. Originally, I had planned for Loop Fusion to be an optimization I applied, but when collecting data for this section, I found that separating the loops I had fused early on in my implementation actually improved performance. More specifically, I saw that the bias, ReLU, and max pooling loops could be fused, as there was no dependency of the convolution layer on the bias layer, so long as the array used to store the sub-tile of the feature map is zero initialized (using `= {{{ 0 }}}` for a 3D array in OpenCL). I fused these three loops, hoping it would improve performance, but once I had found my best-performing version of the program, I removed this optimization to compare the performance before-and-after for this section and found that separating the loops yielded better performance. More interestingly, I found that zero initializing the sub-tile array using `= {{{ 0 }}}` was what was actually degrading my performance. I discovered this because, even when having the loops separated with performance around ~2230 GFlops, if I add the zero initialization and only the zero initialization, then my performance drops to about ~1830 GFlops, which is about the performance achieved when the loops were fused as shown by the table. As to the reason why the difference in performance is so drastic, I’m unsure beyond suspecting that something under-the-hood for zero initialization makes it an expensive operation, or at least one that is considerable enough to hurt the overall execution time when the kernel is executed many times.

The final optimization is memory alignment using `__attribute__((aligned(32 * sizeof(float))))` when initializing the sub-tile array, which aligns the array to 128 bytes, which is the cache-line size stated by the readout for the AWS instance specs when executing `make`. An important caveat, however, is that I only noticed an improvement using this optimization when the sub-tile array was zero-initialized for fusing the bias, ReLU, and max pooling loops. Again, I think this has something to do with the under-the-hood implementation of zero initialization; the results in the table pertain only to a version of my program when Loop Fusion was still applied, but as we can see, memory alignment provides about a ~90 GFlops increase to performance. In my current implementation, this optimization produces a negligible improvement to performance if any, possibly because there are minimal cache misses with the given size of the sub-tile array that is computed for independent channels.

Work Groups vs. Work Items Per Group:

The number of work groups and work items per work group that provides the best performance for my kernel is 1792 work groups and 64 work items per work group, using the parameters `'64 112 16'` and `'4 8 2'` for `OPENCL_WORKGROUP_GLOBAL` and `OPENCL_WORKGROUP_LOCAL` respectively. The Tesla M60 GPU we use has two chips, each with 16 streaming multi-processors (SMs) and 2048 CUDA cores for a total of 32 SMs and 4096 CUDA cores.[2] However, the g3s.xlarge EC2 instance is only allocated one chip, so we have 16 SMs available to us, with 2048 CUDA cores divided amongst them, so each multi-processor has 128 CUDA cores. With 1792 work groups and 64 work items per work group, and 16 multi-processors and 128 CUDA cores per multi-processor, it is evident that these numbers do not match.

The probable reason for why my best performing configuration of work groups versus work items per work group does not match the number of SMs and CUDA cores per SM is likely due to how work is distributed amongst SMs and how we achieve the most parallelization when using GPU acceleration. This is already addressed partly in the **Optimizations** section, but we can consider the case where there are only 16 work groups and 128 work items per work group. Each of these 16 work groups is assigned to a multi-processor as a thread block and the 128 work items in a work group are then divided into groups of 32 as thread warps, wherein each work item is executed by a CUDA core in a group of 32 cores. However, mapping work groups and work items exactly to the number of multi-processors and CUDA cores like this does not yield the best performance, possibly because we don't fully leverage the efficient parallelism that OpenCL and GPU acceleration affords to us. For one, I believe GPUs are efficient in scheduling and distributing work when queuing work items; this is in contrast to a CPU, which has high scheduling overhead, so perfectly mapping work groups and work items to cores and threads does give the best performance as seen in lab 3.[3] However, in the GPU case, we achieve better performance through the divide-and-conquer strategy of parallelization if we divided the problem further into simpler computation units i.e. more work items with less work per work item. This is because we may make better use of the Single Instruction Multiple Data (SIMD) extensions of OpenCL devices and the GPU's SMT architecture, wherein we can have multiple in-flight work items per CUDA core so that we hide memory latency by executing work items, while other work items block when waiting for memory accesses.[4] It is this method of hiding memory latency that is the most probable reason why the best performance is achieved when the numbers of work groups and work items per work group do not match the number of multi-processors and CUDA cores per multi-processor and are actually more than the latter.

Sources Cited:

1. https://stackoverflow.com/questions/12212003/how-concurrent-blocks-can-run-a-single-gpu-streaming-multiprocessor#:~:text=1%20Answer&text=The%20Streaming%20Multiprocessors%20(SM)%20can,process%20akin%20to%20Hypter%2DThreading.&text=The%20execution%20context%20(program%20counters,entire%20lifetime%20of%20the%20warp.

2. https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-maxwell-gpu-accelerators/

3. https://www.hindawi.com/journals/sp/2015/859491/

4. https://stackoverflow.com/questions/50722023/how-does-opencl-distribute-work-items