Vivek Sivakumar
303652195

# Lab 4 Report

*Briefly explain the parallel strategies you applied in this lab. How is it parallelized?*

My general strategy in this lab was to use as many threads as possible to take advantage of the number of GPU cores as well as the low overhead for GPU threads. My first approach was to simply increase the number of work items from 32 to 512 to completely parallelize the outermost for loop, unfortunately this approach still timed-out due to memory access latency.

To solve the memory latency I first created a private Cout array and a private weight array. The private Cout array was the size of the entire Cout array (NUM * IMROW * IMROW) while the private weight array was just the size of one row (KERNEL * KERNEL). This implementation got me a runtime of 11 secs, slower than my lab3 implementation.

After attending the discussion I attempted the suggested approaches of: having one work group per feature map and coalescing memory access (by having a set of n threads work on n pieces of contiguous data). To allow multiple threads to work on the feature map efficiently I had to pull some of the kernel arguments into group-local memory instead of thread-private memory. My initial attempt utilized a local_weight array similar to the previous private array, a local_Cin array that holds (KERNEL * INIMROW) elements, and a local Cout array that holds IMROW elements. This implementation took about 13 secs which was actually worse than my implementation with 512 threads.

After experimenting with this approach some more I found that removing Cout_local and just reading and writing from Cout global improved the performance by almost 7 seconds (down to 6.439 secs). This surprised me quite a bit, as I felt having a local copy of Cout would boost performance. I even tried allocating multiple rows of Cout_local at a time to reduce the number of re-writes on Cout_local, but this approach tanked the performance even further. My takeaway was that allocating large amounts of local memory kills performance and it should be minimized.

Finally I experimented with the number of threads in a work group and found that 128 threads per work group brought my performance to 3.117 secs (any more threads per work group hurt performance). To get to my final implementation I experimented with some more optimizations, including: a private accumulator when calculating Cout updates and allocating local memory in the kernel instead of the host.

*What is the expected communication overhead?*

This program shares much of the communication overhead with lab 3 like allocating buffers for: Cin, weight, bias, and Cout. There is some new overhead that comes from having more than one work group and data in local memory. Each time a work-group makes updates to local memory the work group needs to be synchronized before proceeding any further, this synchronization step requires extra communication between all threads in a work group. I did not add any code that required communication between work groups as this is basically impossible in the GPU and would need to be handled by the host program.

*Evaluate your program in terms of the execution time, and make a comparison with your lab 3.*

My initial implementations of this lab were actually significantly slower than lab 3 which surprised me. However after attending discussion and applying the ideas of having one work group per feature map and coalesced memory access I got the speedup I was hoping for. This just shows the effect that memory latency has on GPU execution times. While the GPU has many more cores, it takes attention to detail and a few tricks to hide the memory latency. My best performance is 2.897 secs which is about a 2x speedup from lab 3. I feel that with a larger problem size the difference would be even more pronounced.

**Summary of Parallelization strategies**

| | |
|---|---|
| Lab 3 Code (32 work-items) | terminated |
| Lab 3 Code (512 work-items) | terminated |
| Private Cout (512 work-items) | 16 secs |
| Private weight and Cout (512 work-items) | 11 secs |
| Local Weight (512 work groups, 32 work-items per group) | 15 secs |
| Local Weight, Cin, & Cout (512 work groups, 32 work-items per group) | 13 secs |
| Local Weight & Cin (512 work-groups, 32 work-items per group) | 6.439 secs |
| Local Weight & Cin (512 work-groups 64 work-items per group) | 3.787 secs |
| Local Weight & Cin (512 work-groups 96 work-items per group) | 3.474 sec |
| Local Weight & Cin (512 work-groups 128 work-items per group) | 3.117 sec |
| Local Weight & Cin, private Cout (512 work-groups 128 work-items per group) | 3.070 sec |
| Local Weight & Cin, private Cout, in-kernel allocation of local variables (512 work-groups 128 work-items per group) | **2.897 sec** |
| Local Weight, Cin (512 work-groups 160 work-items per group) | 3.785 sec |
| Local Weight, Cin (512 work-groups 192 work-items per group) | 4.255 sec |

*Discuss how much memory is used at private, local, and global levels of your best configuration.*

<u>Global</u>

Cout, Cin, weight, bias

Every argument is inputted into the kernel as a variable in global memory.

Local
Cin_local, weight_local

I have a weight_local array that is size (KERNEL * KERNEL). During every iteration of of the j loop I write in that j value's column from weight into weight_local. Then when doing the computation of Cout I can read from weight_local instead of weight global.

I have a Cin_local array that is size (KERNEL * INIMROW). I wanted to just have an array of size INIMROW, but the algorithm can access from rows h, h+1, and h+2 in one iteration of the h loop. In order to satisfy this need I had to bring in the next 3 rows of Cin for every iteration of the h loop. Then when doing the computation of Cout I can read from Cin_local instead of Cin global.

I experimented with a Cout_local (1 row chunks and 28 row chunks) but this just ended up hurting performance.

Private
private_bias, comp

I have a private_bias variable that is set to bias[group_id] for each thread. Then when setting Cout to bias[i], I can use the private_bias instead of doing a read from global bias. By making bias a private variable for all threads I can do a coalesced write to Cout.

I have a comp variable that I use to accumulate the result of the computation section of the algorithm. I then update Cout by adding the comp value to the current Cout value. This eliminates p*q reads of Cout for each thread.

*Challenges you encountered and how you solved them.*

The biggest challenges in this lab were: understanding how to map the algorithm to work-items and work-groups and dealing with memory effectively. At first I thought simply bumping the number of threads would improve performance but I was sadly mistaken, I needed to take some time to understand the best way to divide the threads into groups and how to size those groups to best leverage the underlying architecture.

It actually took less time than I expected to understand the mapping portion, I ended up spending the majority of my time dealing with hiding memory latency. The hardest part for me was manipulating the local memory to work with the algorithm, particularly for Cin. Since the entire Cin doesn't fit in local memory I had to allocate a row at a time, however the algorithm uses more than one row of Cin per iteration. It took me quite awhile to figure out that I had to allocate KERNEL rows of Cin per iteration of the h loop to properly utilize Cin_local.