Computer Science 133, Lab 4

Michael Wu
UID: 404751542

March 11th, 2019

1 Parallelization Strategy

I knew that GPUs work better as the number of parallel items to process increases, so I began by trying to write the smallest kernel possible. I started with my kernel from the previous lab. In the previous lab I wrote the kernel 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 so that each work item corresponded to one output element. 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.

My previous code was unrolled 4 times since this improved performance due to vectorization on the CPU. I undid this optimization so that each work item calculated one output element, since this would be more suited for a GPU. I changed this because GPUs have many more execution lanes than CPUs. Then I tuned the work group size so that the GPU could execute the most number of calculations in parallel. I found that the dimensions $4 \times 1 \times 8$ gave me a speed of about 310 GFlops. I had to run about three times in a row to achieve this, since it seems like I get slower speeds due to a cold cache on the first and second runs.

2 Optimizations

Most of my optimizations were reused from the previous lab. I still calculate multiple elements in parallel in the inner convolution loop and store these into private memory so that the looping is fast. The main differences were that I made the kernel small and tested various work group sizes in order to get the best performance. The work group sizing changed the performance drastically, as I could have speeds as slow as 10 GFlops or as fast as 300GFlops based on my work group size. I also declared my array indexing constants using `__constant` in order to reduce the amount of private memory each work group item uses. I did not measure the effect that this had on my code, but it made it easier to read and may have led to marginal improvements.

3 Work Groups and Work Items

I found that the most optimal number of work groups was $64 \times 112 \times 14$ with $4 \times 1 \times 8$ work items per work group. The specs on the NVIDIA Tesla M60, which is used in the `g3s.xlarge` instance, indicates that there are two GPUs with 2048 cores each. The compute capability of the board is 5.2, so the size of the work group is exactly the warp size which is 32. The total number of work groups, 29120, is much higher than the number of cores. This makes sense since they are distributed among the available cores. When I tried to increase the size of the work groups, I found that I would have errors due to the work groups running out of memory. This is a limitation that prevents us from making the number of work groups equal to the number of cores per GPU.

In order to test various work group sizes, I ran the following bash script that performed grid search on the parameters.

```
#!/bin/sh
export OPENCL_PLATFORM='NVIDIA CUDA'
export OPENCL_DEVICE='Tesla M60'
export OPENCL_SOURCE=nvidia.cl
export OPENCL_WORKGROUP_OFFSET='0 0 0'
export OPENCL_WORKGROUP_GLOBAL='256 112 112'

set -x
```

```
for x in 1 2 4 8 16
do
    for y in 1 2 4
    do
        for z in 1 2 4 7 8 14 16 28
        do
            export OPENCL_WORKGROUP_LOCAL="$x $y $z"
            ./cnn
        done
    done
done
```

I found that increasing the first and third dimensions led to the best performance increases. Increasing the second dimension too much slowed down the program. This is most likely because a small second dimension allows the best cache locality when performing the convolution step, since the computations would access parts of the input array that are close to each other. Here is a subset of my results showing how the speed varies as each parameter is changed.

Dimension 1	Dimension 2	Dimension 3	Speed (GFlops)
4	1	1	57.8425
4	1	8	303.555
4	1	16	302.029
4	2	1	92.9816
4	2	8	271.883
4	2	16	268.107
8	1	1	44.0284
8	1	8	194.669
8	1	16	268.536
8	2	1	85.5747
8	2	8	241.379
8	2	16	244.05

4 Challenges

This lab was pretty straight forward since I reused a lot of the kernel code from the previous lab. When tuning, I had difficulties setting up my grid

search because there were many combinations of work group sizes. Once I had the bash script working I was able to easily choose the work group size that led to the best performance. I was also stuck while thinking about ways to further optimize my code, as it seemed fairly fast already. I intended on trying to use local memory in order to reduce the latency of reading from global memory, but I ran out of time before I could get it working.