Name: *Mohit Mahajan*
NetID: *mohitm3*
Section: *CS483 AL2 (67070)*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.19 ms*	*0.65 ms*	*1.34 s*	*0.86*
1000	*1.65 ms*	*6.23 ms*	*10.051 s*	*0.886*
10000	*16.07 ms*	*62.23*	*1m 38.95 s*	*0.8714*

1. **Optimization 1:** ***Constant Memory and Tiled shared convolution***

    a. Which optimization did you choose to implement and why did you choose that optimization technique.

    - Transferred the weight matrix (k) to the GPU constant memory
    - Because, weight matrix is much smaller in size relative to input matrix in the layer and expect to have faster load for device_k

    - Tiled and shared memory convolution
    - Because, it shall increase the data reuse and effectively utilize memory bandwidth

    b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

    Instead of loading the convolution matrix values from global memory, we load the values in the shared memory first and reuse the value locally to compute partial results.

    We perform tiling in order to partition the overall operation into a parallel compute procedure. Because of the lesser loads from global memory and on top of that loading the weights from constant memory, we expect to have an improved OpTime.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.19 ms*	*0.89 ms*	*1.27 s*	*0.86*
1000	*1.74 ms*	*8.82 ms*	*10.42 s*	*0.886*
10000	*16.83 ms*	*88.05 ms*	*1m 38.4 s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

- Yes it is effective for smaller layer, as visible in the Op time values in the following table
- From the profiling results SOL, at a high level the GPU Utilization (SM) increased from 76% to 83% overall

e. What references did you use when implementing this technique?

Chapter 16, (Wen Mei Hwu), Lecture Notes ECE408

2. **Optimization 2**: ***Input Unrolling and Matrix Multiplication (Separate Kernels and Fusion)***

Note: Separate kernels are intermediate step towards developing the fused kernel, also doesn't perform as good as baseline. Thus, results in this section are recorded for the fused kernel in the report. I still have the two separate implementations In the code.

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Matrix multiplication is a highly parallel and distributable operation, thus if the CNN convolution can be transformed into a matrix multiplication, one can expect to improve the OpTime.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

This optimization operates into two phases, first take a portion of input matrix,

perform the unrolling and then multiply right multiply input matrix with the weight matrix to obtain the convolution. (Cite: Lecture Notes ECE 408 – Lecture 13)

Now, two separate kernels for unrolling and MM still have some opportunity for improvement because, first we load image from the global memory and then the threads perform the transformation and write back to global. Then matrix multiplication kernel in the next phase loads the unrolled values back from global memory into the threads.

However, in the fused kernel, fundamentally it performs the tiled matrix multiplication but it does the unrolling operation while loading the tiles from global memory. Therefore, this process saves a read & write operation to global memory.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).
As noted above: The results are recorded for the fused kernel.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.53 ms*	*0.33 ms*	*1.132 s*	*0.86*
1000	*5.07 ms*	*3.07 ms*	*9.78 s*	*0.886*
10000	*50.46 ms*	*30.446 ms*	*1m 36 s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of)

Yes, as we see for the bigger layer (2) the run time has come down significantly from 62 ms to 30 ms. Following are observations from the Nisght-Compute

- Peak memory utilization increased by 59% relative to baseline
- Global Memory loads decreased by 89.6% relative to baseline
- Stall wait time has decreased, as a result the GPU SM Utilization has increased
- Significantly more active warps per scheduler

e. What references did you use when implementing this technique?

Chapter 16 (Wen Mei Hwu), Lecture Notes

3. **Optimization 3: Loop Unrolling and Matrix Multiplication (compiler based optimizations)**

(Delete this section blank if you did not implement this many optimizations.)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

We do have nested loops ubiquitous in the convolution kernel and over that it might be better to have a thread perform multiple leaps on iterations in one launch expecting reuse of the data from the register memory.

Second, In GPU's the way we have defined the baseline kernel a pointer is not being reused by other SM at the same time, thus we can use __restrict in order to alias pointers while calling the kernel.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

It's a very simple optimization technique that performs the unrolling of the loop before mapping. We leverage the compiler level optimization in this approach. Use #pragma unroll over the thread loops and iterate through several unrolling factors to find the best fit.

Apply restrict argument to the input arrays for pointer aliasing.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.12 ms*	*0.43 ms*	*1.21 s*	*0.86*
1000	*1.12 ms*	*4.15 ms*	*9.503 s*	*0.886*
10000	*10.98 ms*	*42.25 ms*	*1m 33.65*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

e. What references did you use when implementing this technique?

- *https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/*
- *https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=osu1253131903&disposition=inline*
- *https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf*

4. **Optimization 4:** ***FP16 operations***
   (Delete this section blank if you did not implement this many optimizations.)

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      In the hope of improving the compute time by improving the compute time per clock cycle.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

      This optimization perform FP16 floating point operations in the matrix multiplication kernel inspite of the usual FP32 or FP64 floating point operations in the thread. We utilize the "cuda_fp16.h" header to obtain the utility functions for this interop.

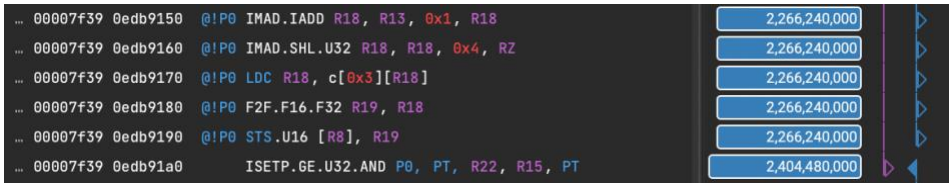
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.30 ms*	*0.74 ms*	*1.15 s*	*0.84*
1000	*2.77 ms*	*7.27 ms*	*9.632 s*	*0.842*
10000	*27.41 ms*	*72.41 ms*	*1m 34.79 s*	*0.849*

   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

      This optimization was not really successful, it led to some accuracy loss (as expected) but at the same time there's a higher runtime.

      As noticed in the nsight compute analysis the time-saved is relatively lesser than what is required for interconversion between the input, compute and output data.



.

e. What references did you use when implementing this technique?

https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

5. **Final Optimization in the submission: Different kernel for different layer sizes and sweeping across parameter values for best performance**

a. Which optimization did you choose to implement and why did you choose that optimization technique.

As seen earlier in the results of different optimization over baselines each of the kernels have their pro's and con's because sometimes kernel is efficient but there's a fixed cost to pay (as in unrolled matrix multiplication) on the other side sometimes kernel is fast but not scalable. Thus picking up the best component mixture is the objective of this optimization.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

Use tiled shared memory kernel for smaller layer
Use input unrolling and matrix multiplication for bigger layer

- The reasoning is based on the performance as seen in the above optimizations

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.19 ms*	*0.33 ms*	*1.143 s*	*0.86*
1000	*1.70 ms*	*3.05 ms*	*10.13 s*	*0.886*
10000	*16.81 ms*	*30.43 ms*	*1m 42.9 s*	*0.8714*