Name: *William Cheng*
NetID: *shihuac2*
Section: *AL2*

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	*0.308775ms*	*1.08427ms*	*1.184s*	*0.86*
1000	*2.91995ms*	*10.5655ms*	*9.62s*	*0.886*
10000	*28.8746ms*	*105.843ms*	*1m35.424s*	*0.8714*

1. **Optimization 1: *Tiled shared memory convolution (2 points)***

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

    I chose to implement the tiled shared memory convolution because there are multiple memory accesses to each input element during convolution, but the global memory bandwidth is limited. Also, reading from global memory is slower than reading from shared memory. Using the tiled shared memory method decreases the number of global memory reads and therefore increases performance.

    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?

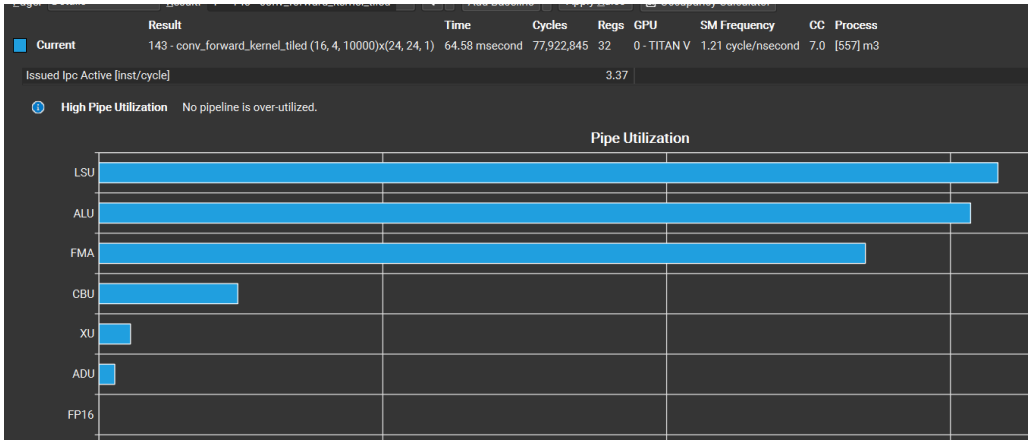
    Strategy 2 from lecture 8 is used for this optimization. Namely, each block covers a tile of the input tensor, loads every element of the input tile into the shared memory, and calculates the values for a smaller output tile. The algorithm is the same as the baseline implementation except for some coordinate shift calculations. I thought the optimization would increase the performance of the forward convolution since it reduces the number of global memory accesses by reusing the values in the shared memory. Reading from global memory can be a bottleneck for the baseline implementation since the memory bandwidth is limited and global memory reads are not as fast as shared memory reads. Using tiled shared memory avoids this problem.

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	*0.314426ms*	*0.672168ms*	*1.154s*	*0.86*
1000	*2.98382ms*	*7.05919ms*	*9.618s*	*0.886*
10000	*29.5441ms*	*64.5782ms*	*1m35.358s*	*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, this optimization was successful in improving performance. The total OpTime is improved from 134.7176ms in the baseline implementation to 94.1223ms. The reasons are stated in (b). Following is the Nsight compute profile:



The execution time of the second conv layer is 64.58ms, which is faster than the baseline's 105.843ms. The most utilized units are LSU, ALU and FMA. LSU is for global memory access, ALU is for integer calculation and FMA is for FP32 operations. The performance increase should be due to lower LSU utilization. The performance can be further improved by converting FP32 to FP16, so that the FMA unit will be replaced by the FP16 unit which is much faster.

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

The notes from Lecture 8

2. **Optimization 2:** ***FP16 arithmetic + multiple kernel implementations for different layer sizes + kernel in constant memory (6 points total)***
    a. Which optimization did you choose to implement and why did you choose that optimization technique.

    *I chose to implement FP16 arithmetic operations in the convolution kernel along with multiple kernel implementations and kernel in constant memory. The reason for choosing FP16 is that it is much faster than FP32 operation previously used. Each thread calculates Channel*K*K floating point multiplications, so using FP16 is likely to increase the performance by a significant amount.*

    The reason for choosing multiple kernel implementations is that the two convolution layers have different configurations that can be optimized. For Layer 1, it only has one input channel, so I chose to use the half type for calculation. For Layer 2, it has four input channels. I chose to use __half2 to store a pair of values for two different channels. As a result, each thread only has to iterate 2 times for each channel instead of 4 times in the original implementation. Additionally, the multiplication operation function used is __hmul2, which is about the same execution time as single float type multiplication, thanks to the hardware acceleration. Half of the original iterations along with FP16 will increase the performance by more than 2x.

    The reason for storing kernels in constant memory is to decrease read latency when the kernel is accessing the mask values. Additionally, the masks stored in constant memory are pre-processed by the CPU so that the values are converted from float32 to half and __half2. This will slow down the whole program execution but will further increase kernel performance since the kernel doesn't have to convert the mask values upon each access.

    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?

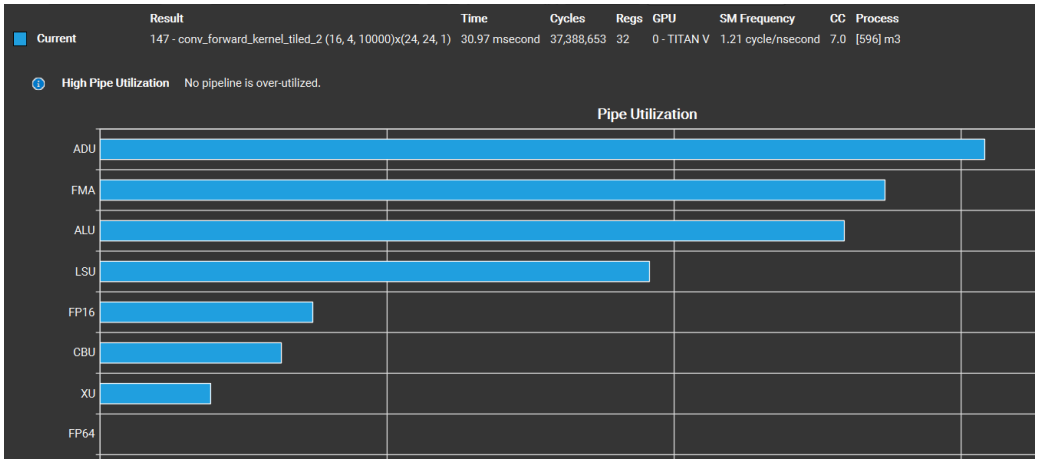
    The optimization works by converting FP32 multiplications to __hmul (half) and __hmul2(__half2). Both these operations are faster. The mask values are converted to half precision and stored in the constant memory. I thought it would slightly improve the first convolution layer, and vastly improve the second convolution layer. The reasons have been stated above. It ended up boosting the total OpTime to 56.2874ms. This optimization does synergize with the previous optimization, which is the tiled shared memory convolution.

    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	*0.469287ms*	*0.522356ms*	*1.202s*	*0.86*
1000	*2.71169ms*	*3.3299ms*	*9.592s*	*0.887*
10000	*25.1125ms*	*31.1749ms*	*1m33.943s*	*0.8715*

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, this optimization was successful in improving performance. This is mainly due to the high performance of FP16 operations, the details have been stated in (a) and (b). Following is the Nsight compute profile:



The execution time of the second conv layer kernel is 30.97ms, which is around 2x faster than the previous optimization. The FMA here does not do FP32 operations, but __hmul2 operations on __half2 type. The FP16 unit is used for __hadd operation since the final output value is the sum of both halves of the accumulated __half2 value. LSU utilization significantly decreased because constant memory is used for the mask. ADU is high due to constant memory reads. The performance boost is mainly due to the

__hmul2 operation, which is the same performance as a single FP32 multiplication but calculates two values instead of one, resulting in a ~2x performance boost.

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

For constant memory and multiple kernel implementation, I used the textbook and lecture notes.
For FP16 arithmetic and conversion functions, I used the Nvidia CUDA toolkit documentation:
https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html#group__CUDA__MATH__INTRINSIC__HALF

3. **Optimization 3:** ***Using streams to overlap computation with data transfer (4 points)***

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

I chose to implement using streams to overlap computation with data transfer. I chose this optimization because my previous optimization has already achieved a ~56ms total OpTime which is good enough, so I want to do some optimizations on the total execution time.

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?

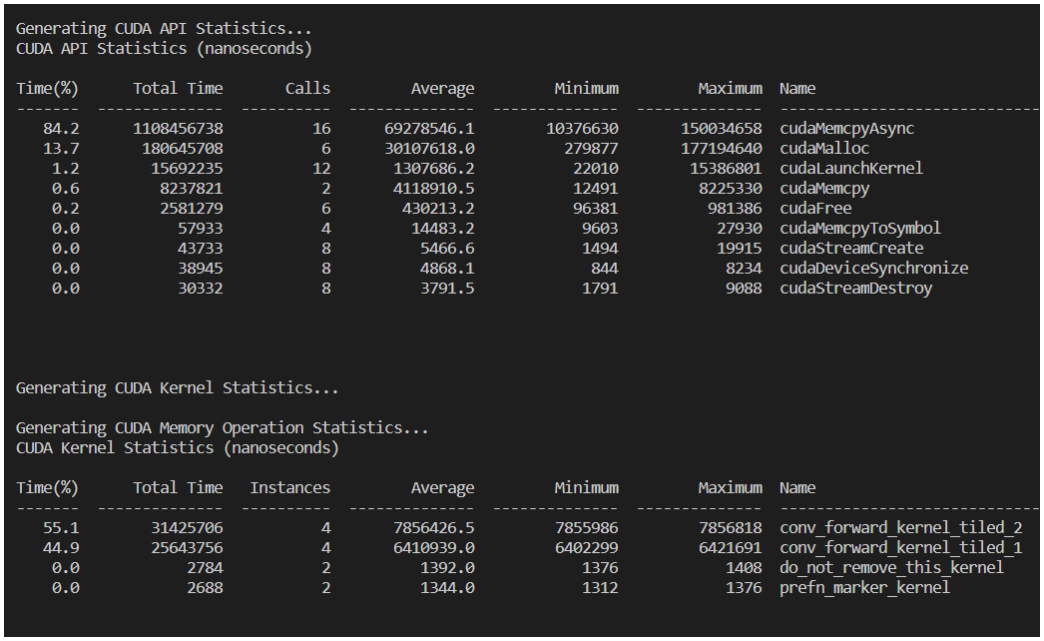
In my implementation, I used 4 streams to divide the whole dataset. The z-dimension of the grid size (which is Batch) is set to Batch/4, block size is left unchanged. On the CPU side, the conv_forward_gpu does 4 iterations. In each iteration, ¼ of the input data is transferred to the device memory, then the corresponding stream executes the kernel, and finally the output data is transferred from device to host. I did not think the optimization would increase performance of the convolution since the modifications are mainly on the CPU side. It is for more efficient data transfer instead of a faster kernel. It should synergize with the previous optimization, but not in terms of kernel execution speed, but data transfer efficiency.

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	*9.06921ms*	*7.76622ms*	*1.319s*	*0.86*
1000	*73.9799ms*	*56.8598ms*	*10.153s*	*0.887*
10000	*629.122ms*	*468.252ms*	*1m33.739s*	*0.8715*

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 did not improve the kernel performance. Following is a screenshot of the nsys profile:



```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   84.2      1108456738          16      69278546.1        10376630       150034658  cudaMemcpyAsync
   13.7       180645708           6      30107618.0          279877       177194640  cudaMalloc
    1.2        15692235          12       1307686.2           22010        15386801  cudaLaunchKernel
    0.6         8237821           2       4118910.5           12491         8225330  cudaMemcpy
    0.2         2581279           6        430213.2           96381          981386  cudaFree
    0.0           57933           4         14483.2            9603           27930  cudaMemcpyToSymbol
    0.0           43733           8          5466.6            1494           19915  cudaStreamCreate
    0.0           38945           8          4868.1             844            8234  cudaDeviceSynchronize
    0.0           30332           8          3791.5            1791            9088  cudaStreamDestroy



Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   55.1        31425706           4       7856426.5         7855986         7856818  conv_forward_kernel_tiled_2
   44.9        25643756           4       6410939.0         6402299         6421691  conv_forward_kernel_tiled_1
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2688           2          1344.0            1312            1376  prefn_marker_kernel
```

According to the output of rai when directly running "./m3", the OpTimes are much higher than the baseline implementation. This is because I put all the cudaMemcpyAsync functions inside the conv_forward_gpu function. As a result, the OpTimes represent the sum of the data transfer time and the kernel execution time. In Nsys, as shown in the screenshot, the total kernel execution time is separated from the data transfer time:

- *OpTime1: 25.6438 ms*
- *OpTime2: 31.4257 ms*

This is almost identical to the previous optimization. This is not surprising because I did not make any changes to the kernel. The modifications are all on the host side. The kernel I used is from the previous optimization. The total execution time of the program is also similar to the previous optimization. An advantage of this optimization is that the output data is transferred from the device during the program execution, so it is faster for the data to be ready to use.

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

Notes from lecture 22