Name: *Zhongbo Zhu*
NetID: *zhongbo2*
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	*0.176356 ms*	*0.635522 ms*	*0m1.208s*	*0.86*
1000	*1.63119 ms*	*6.29065 ms*	*0m11.492s*	*0.886*
10000	*16.4997 ms*	*59.9981 ms*	*1m45.378s*	*0.8714*

1. **Optimization 1: Kernel in constant memory**

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

    Weight matrix (kernel values) in constant memory (1pt) because the kernel is accessed often, and I don't want the GPU to access global memory every 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?

The optimization aims to improve the memory access speed for reading data from kernel, which should be constant.

I think it will increase the performance a little, because the memory access to constant memory will be cached more since read-only kernel is cache safe.

This is the first optimization I tried.

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	*0.162441 ms*	*0.579635 ms*	*0m1.073s*	*0.86*
1000	*1.49912 ms*	*5.72161 ms*	*0m11.821s*	*0.886*
10000	*13.225 ms*	*51.5309 ms*	*1m39.210s*	*0.8714*

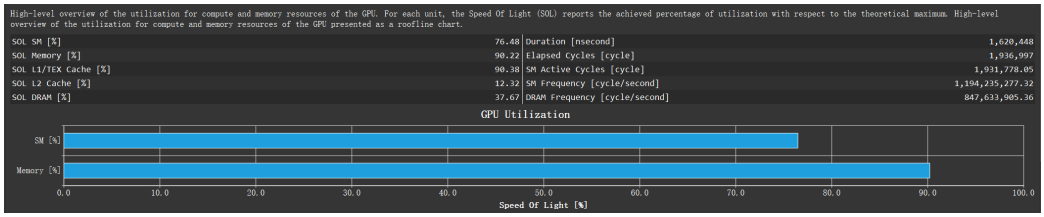
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

The performance is increased because the op time get reduced thanks to the constant memory kernel.

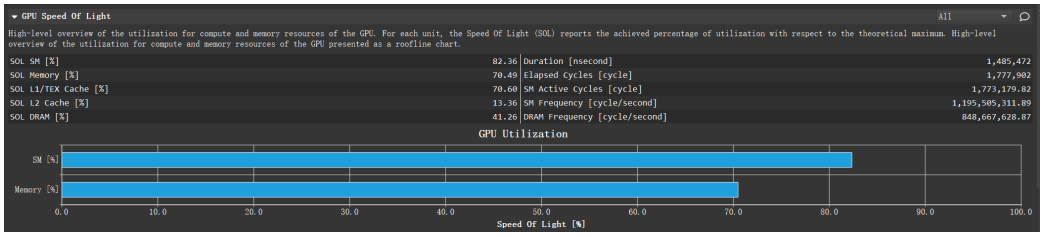
If we compare on batch size 1000.

Nsight-Compute:

Baseline:



New:



As you can see, the SM utilization is higher, and the memory bandwidth utilization is also lower, meaning that we save more memory accessing time, release the pressure added to memory lanes and the SM gets more data to compute.

Nsys:

Baseline:

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	7896519	2	3948259.5	1619477	6277042	conv_forward_kernel
0.0	2848	2	1424.0	1408	1440	do_not_remove_this_kernel
0.0	2752	2	1376.0	1344	1408	prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
61.2	101711876	2	50855938.0	42435599	59276277	[CUDA memcpy DtoH]
38.8	64520240	8	8065030.0	1184	30365829	[CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
226153.0	8	28269.0	0.004	100000.0	[CUDA memcpy HtoD]
172250.0	2	86125.0	72250.000	100000.0	[CUDA memcpy DtoH]

New:

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	7228327	2	3614163.5	1510548	5717779	conv_forward_kernel
0.0	2688	2	1344.0	1312	1376	prefn_marker_kernel
0.0	2656	2	1328.0	1280	1376	do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
64.7	97272136	2	48636068.0	41390973	55881163	[CUDA memcpy DtoH]
35.3	53166850	8	6645856.3	1504	23159755	[CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
226153.0	8	28269.0	0.004	100000.0	[CUDA memcpy HtoD]
172250.0	2	86125.0	72250.000	100000.0	[CUDA memcpy DtoH]

As you can see, the kernel time drop from 7896519 to 7228327. We are saving kernel execution time.

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

The lecture slides.

2. **Optimization 2: Shared memory in conv layer**

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

      Tiled shared memory convolution (2pt).
      I chose it because I have already done the tiling mechanism in the baseline, it would be great that if the memory access to the tile in shared memory.

   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?

      By first loading input to shared memory before doing calculation in kernel, we can make memory access faster.

      I think the optimization will work because the global memory is stilled accessed too often, and we can optimize it.

      This optimization is based on the previous one with constant memory.

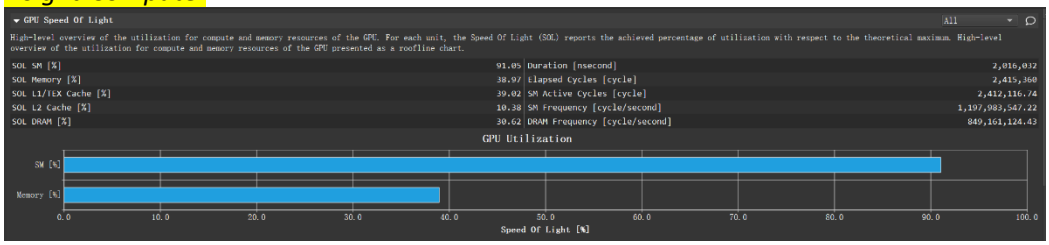
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.310116 ms*	*1.01285 ms*	*0m1.059s*	*0.86*
1000	*2.02896 ms*	*9.76816 ms*	*0m11.856s*	*0.886*
10000	*17.986 ms*	*90.566 ms*	*1m39.410s*	*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).

The optimization was not able to improve based on the previous one, because

Nsight-Compute:



The SM usage is higher, so the share memory is indeed releasing the workload of global memory, but maybe the increase of SM usage is not enough for the extra cost of copying data into shared memory.

Nsys:

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances        Average        Minimum        Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------
  100.0        11928055           2       5964027.5         2053934         9874121  conv_forward_kernel_share
    0.0            2848           2          1424.0            1344            1504  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1280            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations        Average        Minimum        Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------
   65.4       107580362           2      53790181.0        43672126        63908236  [CUDA memcpy DtoH]
   34.6        56966474           8       7120809.3            1472        24242346  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

          Total      Operations        Average        Minimum        Maximum  Name
----------------  --------------  --------------  --------------  --------------  ----------------------------------------------
       226153.0               8         28269.0           0.004        100000.0  [CUDA memcpy HtoD]
       172250.0               2         86125.0       72250.000        100000.0  [CUDA memcpy DtoH]
```

You can see that there is an increase of kernel execution time.

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

The lecture slides.

3. **Optimization 3: Matrix multiplication and input unrolling with share memory**

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

Shared memory matrix multiplication and input matrix unrolling (3 points).

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 is basically converting convolution to matrix multiplication, in which we can make the memory access more coalesced, and hopefully increase the performance. That's why I think this optimization will work.

This optimization is still using the constant memory, but the previous one using shared memory in conv kernel is not used.

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).

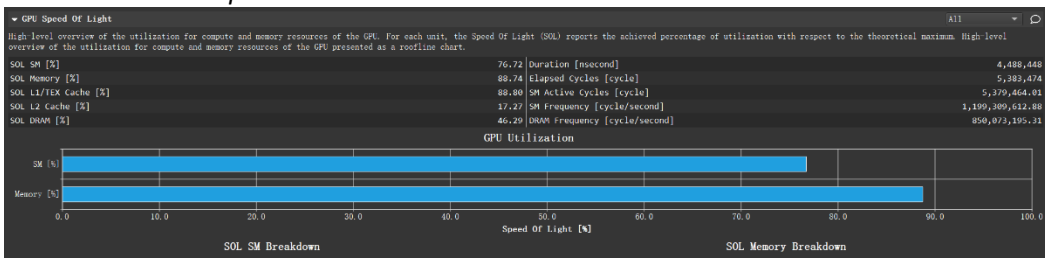
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.13229 ms*	*1.53565 ms*	*0m1.062s*	*0.86*
1000	*10.0351 ms*	*14.5446 ms*	*0m12.073s*	*0.886*
10000	*100.231ms*	*120.324*	*1m39.460s*	*0.8714*

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).

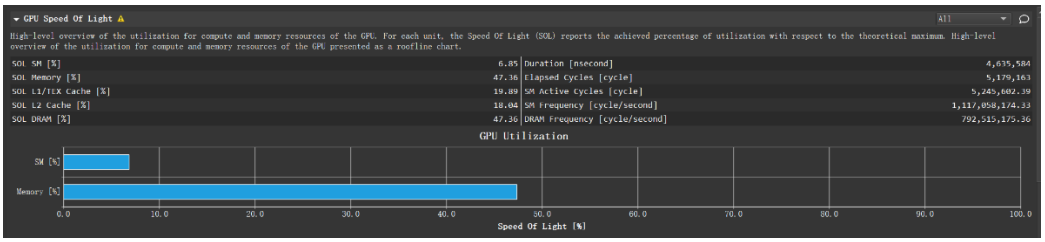
The performance of this optimization is not working with better performance, because it may take too much time to launch two kernels (input unroll and matrix multiplication), and there are replicated elements in the matrix we unrolled, meaning extra calculation needed. So the time saved for memory accessing cannot compensate that part of extra time.

Nsight-Compute:

This is matrix multiplication kernel:



This is the preparation unrolling kernel:



As you can see, the algorithm for unrolling is not specially optimized, and it's very time consuming with relatively low SM usage. The matmul kernel is taking too much of the memory bandwidth even with the shared memory. That's the reason why it's not very efficient as expected.

Nsys:

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
51.7	11832666	2	5916333.0	4546265	7286401	conv_forward_kernel_matmul
48.3	11047135	2	5523567.5	4387514	6659621	conv_forward_unroll
0.0	2656	2	1328.0	1312	1344	prefn_marker_kernel
0.0	2656	2	1328.0	1312	1344	do_not_remove_this_kernel

As you can see that the two kernels take a lot of extra work here and greatly slows down the program.

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

The lecture slides.

4. **Optimization 4: Stream**

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

Pick the best strategy from above, apply stream to it so that we can overlap the kernel execution time and the memcpy time.

I think the correct implementation of steam will have great impact since memcpy time is the major cost here.

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 is to open multiple streams, each copying data inside kernel, and invoke the kernel execution. So when the previous kernel call is still executing, I can copy data inside. That's why I think it will improve the performance.

This optimization is based on the baseline implementation + constant memory of kernel.

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).

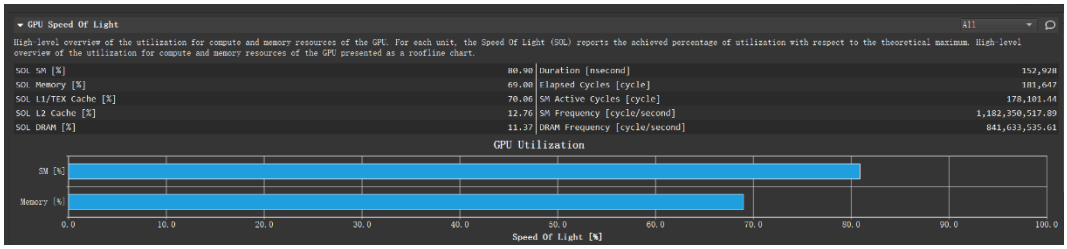
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*6.20684 ms*	*5.52351 ms*	*0m1.108s*	*0.86*
1000	*58.2814 ms*	*44.2621 ms*	*0m11.450s*	*0.886*
10000	*610.649 ms*	*454.014 ms*	*1m40.233s*	*0.8714*

As you can see, the op time here is very large, but that's because the op time metric cannot really capture the "real op time" since we are using the stream. The overall layer is nearly the same with op time.

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 implementation is effective, even though it cannot be reflected in the op time, but you can see a drop in the layer time, since the pipelining method of overlapping the kernel execution with memory copy.

Nsight-Compute:



There are many kernels in the nsight output because we are using stream, nearly all of them are using more than 80% of the SM, but still with relatively low pressure on memory bandwidth.

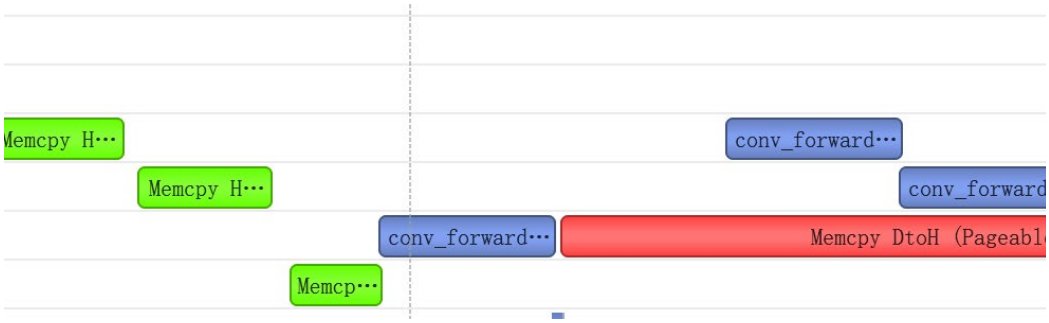
Nsys:

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	7437194	20	371859.7	155647	593115	conv_forward_kernel
0.0	2720	2	1360.0	1344	1376	prefn_marker_kernel
0.0	2624	2	1312.0	1248	1376	do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
90.1	81404241	20	4070212.0	3479559	4687934	[CUDA memcpy DtoH]
9.9	8982847	24	374285.3	1504	482780	[CUDA memcpy HtoD]



As you can see, the we create some overlap between copy in, copy out and kernel execution.

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

The lecture slides.

5. **Optimization 5: Kernel fusion for matrix multiplication kernel**

   a. Which optimization did you choose to implement and why did you choose that optimization technique.
   Kernel fusion for unrolling and matrix-multiplication (requires previous optimization) (2 points)

   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 aims at reducing the scheduling, invocation overhead by using only one kernel.

   What's more, we no longer need to allocate extra space to convert the input x to x_unroll, but instead, we choose to achieve the unrolling by advanced indexing in the kernel execution, so we save huge amount of data copy. That's why I am expecting it be beat the baseline.

   This optimization is using input unrolling, constant memory, and matrix multiplication for conv with shared memory.

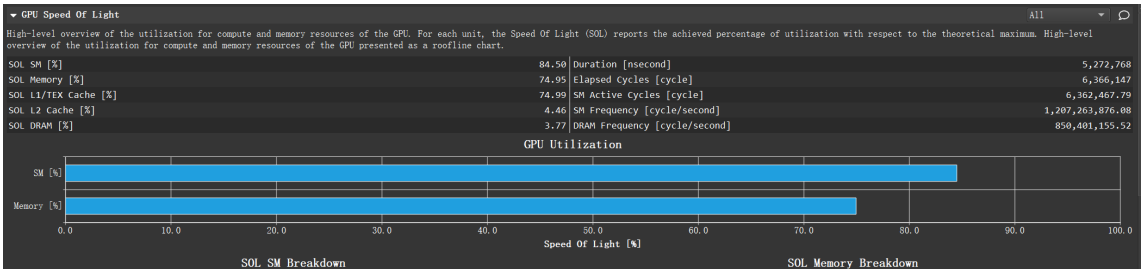
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.544055 ms*	*0.734371 ms*	*0m1.131s*	*0.86*
1000	*5.30461 ms*	*3.22117 ms*	*0m10.124s*	*0.886*
10000	*52.6996 ms*	*72.5554 ms*	*1m36.722s*	*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).

The performance is still not the best, but much better than the previous matmul solution.

According to Nsignt-Compute:



Here, the memory access workload and SM utilization is pretty similar to the baseline + constant memory, much better than the version without fusion. That's why its result is pretty near 70ms. I think the reason why it cannot go further is that the logic inside this kernel function is more complicated than our best version, which might have bigger impact in the overall performance.

Nsys:

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	12614117	2	6307058.5	5358169	7255948	conv_forward_kernel_matmul_fusion
0.0	2752	2	1376.0	1344	1408	prefn_marker_kernel
0.0	2528	2	1264.0	1216	1312	do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
92.8	92693790	2	46346895.0	38111659	54582131	[CUDA memcpy DtoH]
7.2	7206219	6	1201036.5	1472	3898531	[CUDA memcpy HtoD]

The cudaMemcpy time is significantly smaller than the previous matmul, and it's the same with the baseline (since we don't have extra memcpy). That's why the performance is still better than previous matmul. However, the total execution time is still lower than our baseline + constant memory, and maybe it's because the time saved by shared memory cannot compensate the cost of loading time from high dimensional data array (note that the x has batch=10000),so the process of loading x into shared memory might result in huge amount of global memory access (not coalesced).

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