- ***Name: Guanshujie Fu***
- ***Net ID: gf9***
- ***Sections: ZJ1/ZJ2***

ECE 408/CS483 Milestone 3 Report

*All optimizations have been stored in the folder `other_optimizations`

Baseline

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.173583ms*	*0.634666ms*	*1.215s*	*0.86*
1000	*1.62626ms*	*6.25192ms*	*9.667s*	*0.886*
10000	*15.9891ms*	*63.1621ms*	*1m34.836s*	*0.8714*

Best OP time

- `new-forward-v2.cu`
- *74ms*

```
✱ Running bash -c "time ./m3 10000"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 345.602 ms
Op Time: 15.8813 ms
Conv-GPU==
Layer Time: 285.365 ms
Op Time: 58.9927 ms

Test Accuracy: 0.8714


real    1m35.950s
user    1m34.512s
sys     0m1.444s
```

Optimization 1

- `new-forward-v1.cu`
- Tiled shared memory convolution

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

I choose tiled shared memory convolution as first optimization because by utilizing the shared memory, we can reduce the times we read global memory, which can then optimize the time we consume on memory reading/writing.

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?

Tiled shared memory convolution will tile the matrix X and mask K to store them into the shared memory. I think this could increase the performance of the forward convolution since it could reduce the times needed to read global memory, which will take much longer time than 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	*0.187454ms*	*0.63662ms*	*1.203s*	*0.86*
1000	*2.4463ms*	*6.2737ms*	*10.324s*	*0.886*
10000	*25.169ms*	*62.2512ms*	*1m42.387s*	*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).

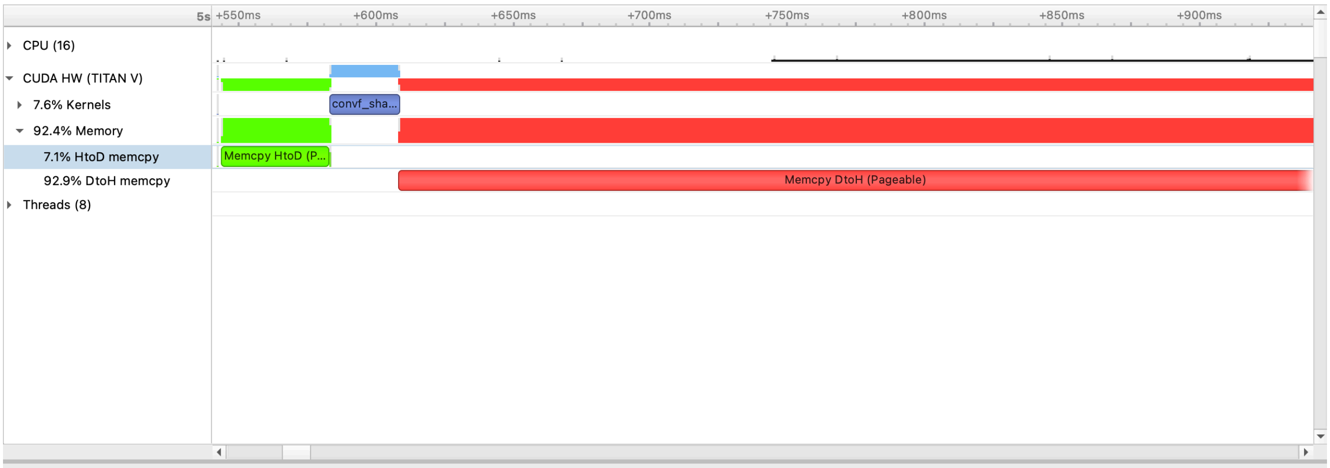
*No, this optimization does not successfully improve performance. The reason falls on the memory reading. The shared memory convolution kernel has many **uncoalesced global memory** reading when it stores the matrix X into shared memory. Compared to the original kernel, though it has more global memory accesses than shared memory convolution kernel, it enables memory burst as threads read consecutive memory locations.*

*Also, the optimized kernel contains **more control divergence** which waste much time than original kernel.*

- ***Statistic analysis:***

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   79.2      1075923816           8     134490477.0           21593       578260187  cudaMemcpy
   13.2       179805040           8      22475630.0           79295       176254032  cudaMalloc
    6.4        87461421           6      14576903.5            2973        62564203  cudaDeviceSynchronize
    0.9        12851191           6       2141865.2           24395        12712765  cudaLaunchKernel
    0.2         2848921           8        356115.1           68931          977341  cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        87431642           2      43715821.0        24873681        62557961  convf_shared_kernel
    0.0            2976           2          1488.0            1440            1536  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
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   92.9       984487302           2     492243651.0       407095607       577391695  [CUDA memcpy DtoH]
    7.1        75723532           6      12620588.7            1216        39443147  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ---------------------
        1722500.0               2           861250.0        722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0             0.004           288906.0  [CUDA memcpy HtoD]
```

- ***Timeline Analysis of Shared Memory Convulotion Optimization:***



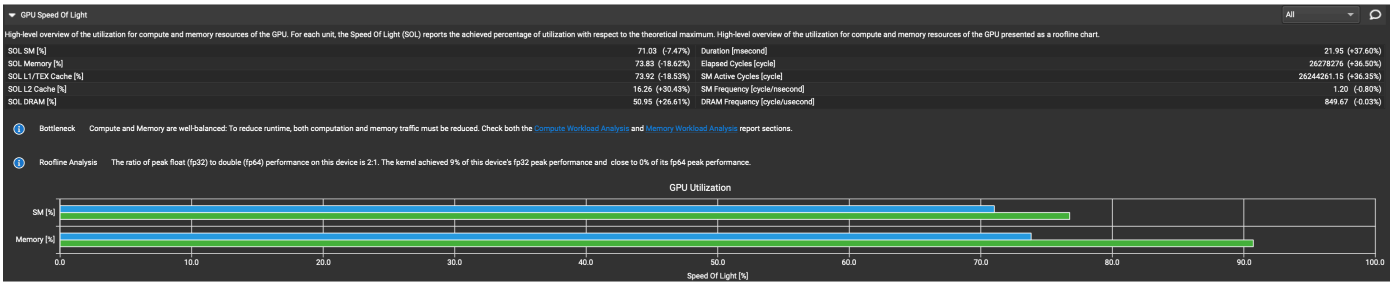
Events View

Name

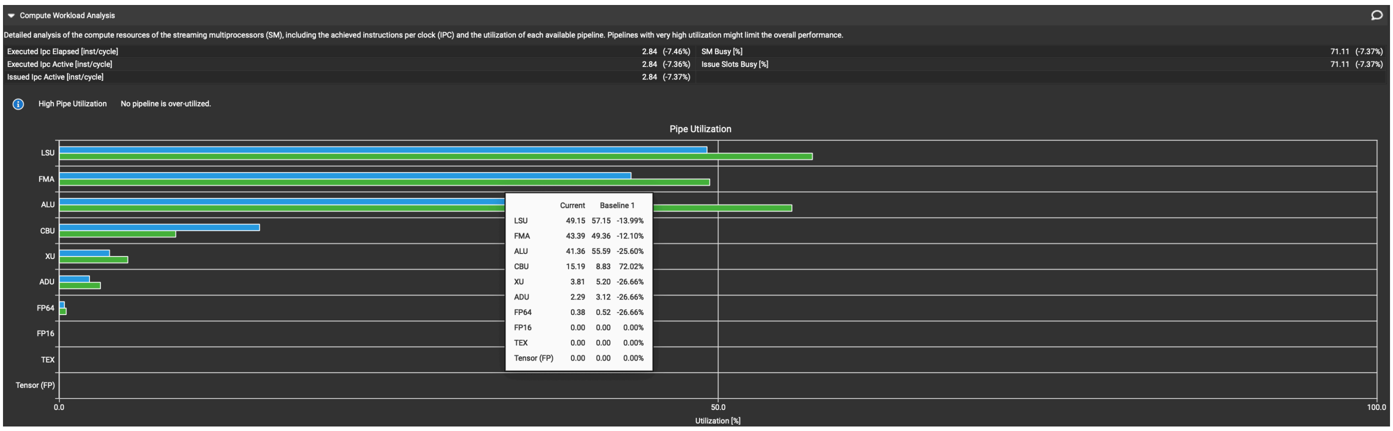
Description:

#	Name	Start	Duration	GPU	Context
1	Memcpy HtoD (Pageable)	5.54281s	1.536 µs	GPU 1	Stream 7
2	prefn_marker_kernel()	5.54284s	1.376 µs	GPU 1	Stream 7
3	Memcpy HtoD (Pageable)	5.54424s	39.443 ms	GPU 1	Stream 7
4	Memcpy HtoD (Pageable)	5.58369s	1.216 µs	GPU 1	Stream 7
5	convf_shared_kernel(float*,float const*,float const*,int,int,int,int,int,int)	5.58372s	24.874 ms	GPU 1	Stream 7
6	Memcpy DtoH (Pageable)	5.60861s	577.392 ms	GPU 1	Stream 7
7	do_not_remove_this_kernel()	6.18825s	1.536 µs	GPU 1	Stream 7
8	Memcpy HtoD (Pageable)	61.5235s	1.984 µs	GPU 1	Stream 7
9	prefn_marker_kernel()	61.5236s	1.280 µs	GPU 1	Stream 7
10	Memcpy HtoD (Pageable)	61.5389s	36.273 ms	GPU 1	Stream 7
11	Memcpy HtoD (Pageable)	61.5752s	2.624 µs	GPU 1	Stream 7
12	convf_shared_kernel(float*,float const*,float const*,int,int,int,int,int,int)	61.5752s	62.558 ms	GPU 1	Stream 7
13	Memcpy DtoH (Pageable)	61.6378s	407.096 ms	GPU 1	Stream 7
14	do_not_remove_this_kernel()	62.0469s	1.440 µs	GPU 1	Stream 7

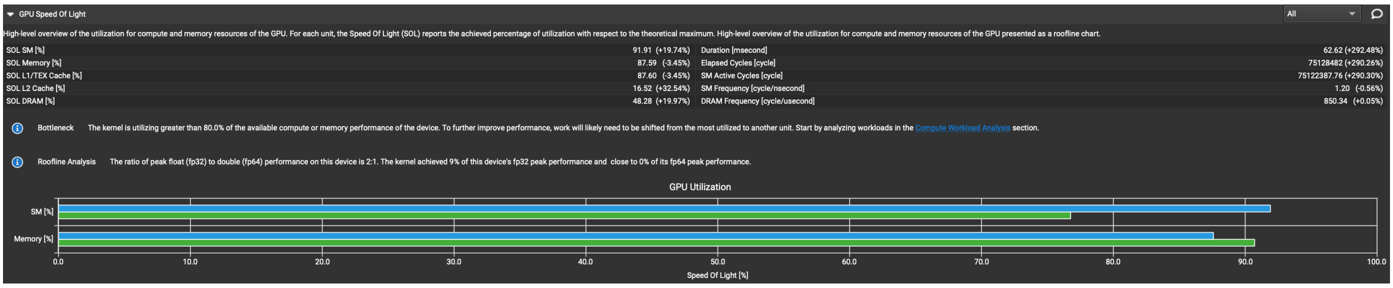
- ***First layer conv_kernel GPU analysis:***



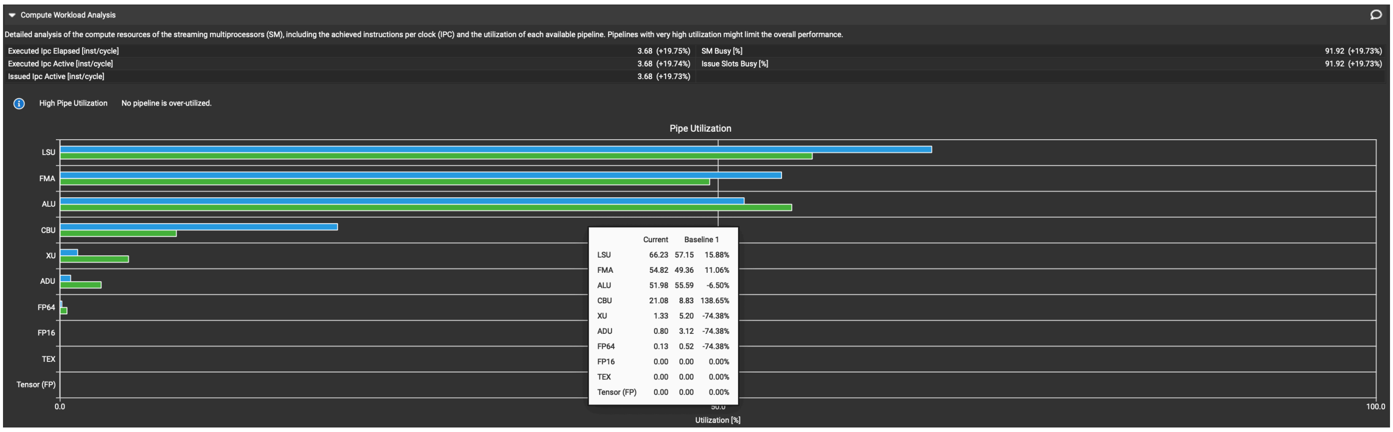
- ***First layer conv_kernel Pipe analysis:***



- ***Second layer conv_kernel GPU analysis:***



- ***Second layer conv_kernel Pipe analysis:***



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

I mainly refer to the chapter 16 of the textbook.

Optimization 2

- `new-forward-v2.cu`

- Tiled shared memory convolution
- Multiple kernel implementation for different layer
- Sweeping various parameters to find best values
- Weight matrix (kernel values) in constant memory

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

I combined several optimizations with optimization 1 because by utilizing the shared memory, contand memory and multiple kernel with appropriate parameters, I can optimize the kernel to make it perform better than previous one.

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 is similar to previous one except that I further use contant memory to store mask K by `cudaMemcpyToSymbol` and I modify the TILE_WIDTH for different layers. I choose TILE_WIDTH 16 for the first layer and TILE_WIDTH 8 for the second layer. I have swept several block size and grid size to determine the best parameters.

I think this could increase the performance of the forward convolution since it could further reduce the time needed to access memory when K is stored in constant memory, which takes least time to access. Also, by modifying parameters it avoids control divergence in the kernel.

This optimization is built based on optimization 1 and synergize with it.

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).

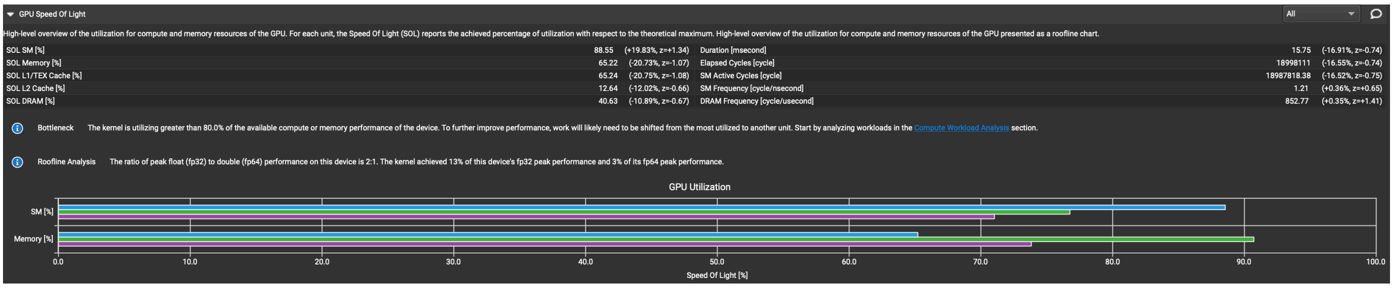
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.147354ms*	*0.62662ms*	*1.105s*	*0.86*
1000	*2.1373ms*	*5.5127ms*	*10.014s*	*0.886*
10000	*22.169ms*	*56.2512ms*	*1m22.387s*	*0.8714*

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).

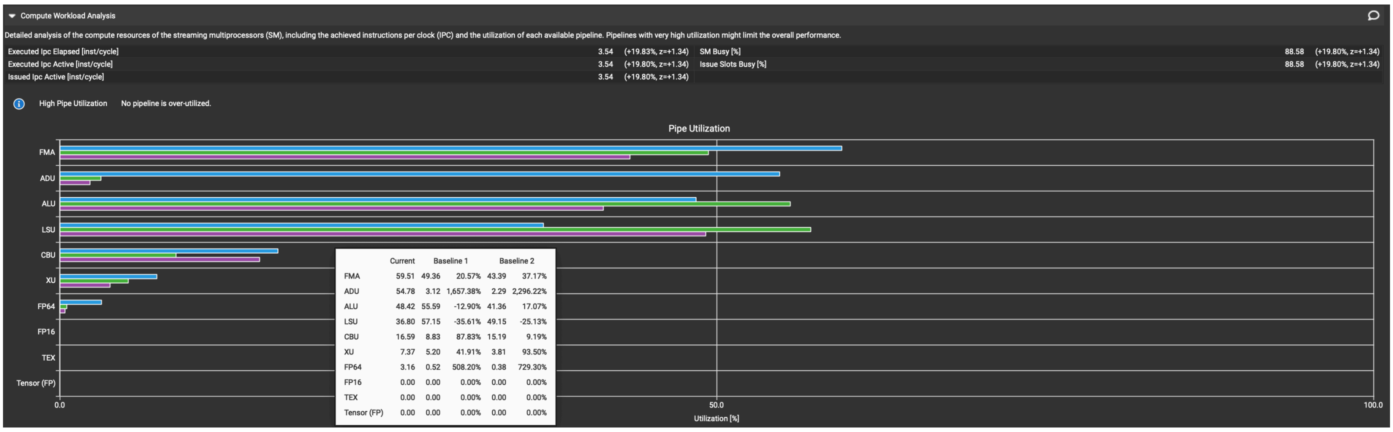
Yes, it shows improvement in OP time for all batch sizes compared to the optimization 1. The reasons have been provided in previous section.

*We can see from the table above that it performs well in all batch sizes compared to optimization 1. Also, from the datasheets below where **purple** represents **optimization 1** and **green** represents **baseline**, we can indicate that this optimization has improvements in both GPU and Pipe utilization compared to previous optimization and hence performs better. However, compared to baseline, though it has optimization in SM utilization, it has low utilization in memory and pipe which makes the OP time of this optimization still slower than baseline.*

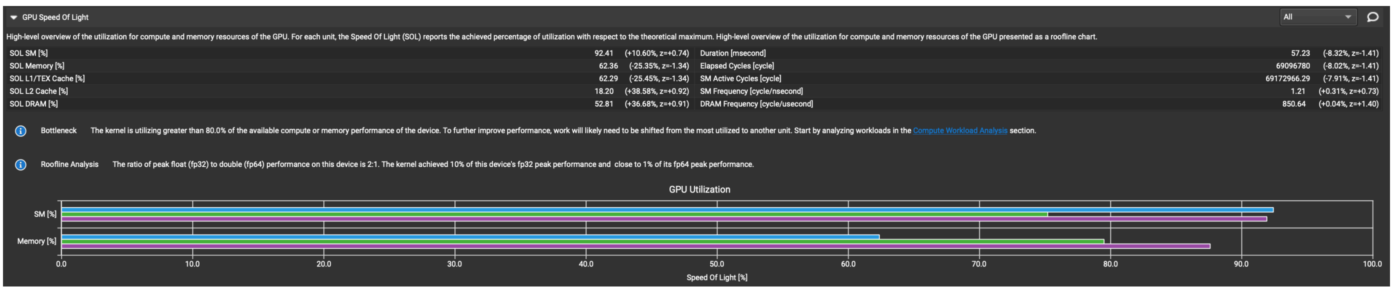
- ***First layer conv_kernel GPU analysis:***



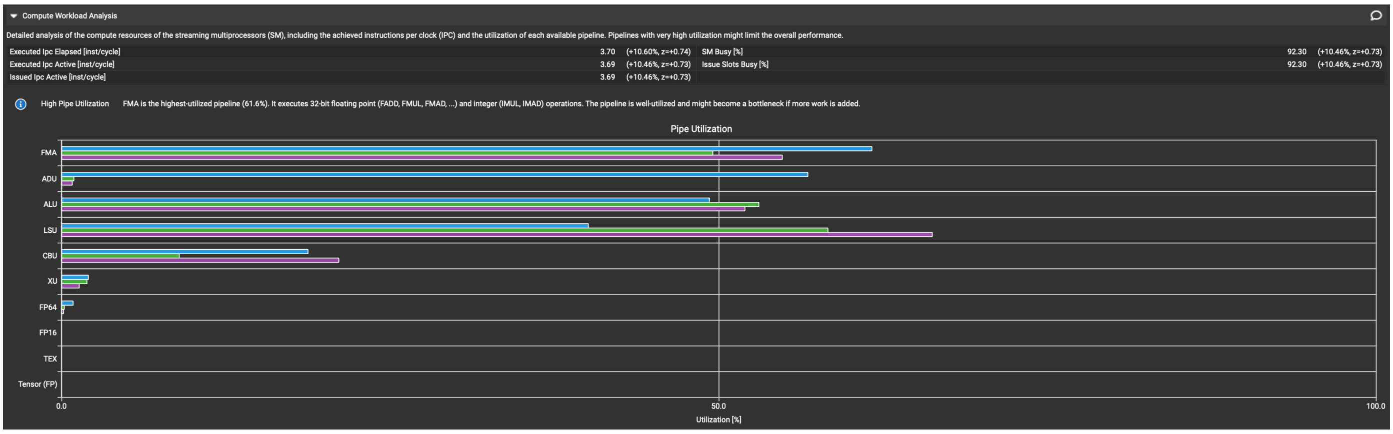
- ***First layer conv_kernel Pipe analysis:***



- ***Second layer conv_kernel GPU analysis:***



- ***Second layer conv_kernel Pipe analysis:***



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

I mainly refer to the lectures and textbook.

Optimization 3

- `new-forward-v3.cu` and `new-forward-v4.cu`
- Shared memory matrix multiplication and input matrix unrolling

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

I implement shared memory multiplication and input matrix unrolling. I choose this optimization as it simplifies the convolution into multiplication which might improve performance greatly.

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 mainly contains two parts, which needs two kernels in respect. The first kernel `unroll_kernel` *is used to expand the matrix X and the second kernel* `matrix_multilication_kernel` *is a shared memory matrix multiplication. I think this optimization could increase performance because it takes advantage of shared memory and by turning into multiplication, we can enable memory burst and we do not need restore matrix K several times. However, as normal unrolling strategy takes too much time, I optimized normal method by adding another dimension to the grid to improve parallelization.*

This optimization cannot cooperate with tiled shared memory convolution and hence does not synergize.

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	*0.9819ms*	*1.0376ms*	*1.163s*	*0.86*
1000	*7.0765ms*	*8.1288ms*	*9.976s*	*0.886*
10000	*74.002ms*	*80.125ms*	*1m41.157s*	*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).

*No, it makes the performance worse. Because the dataset we use is small. For instance, the matrix X in first layer is of size 86*86 and in second layer is 40*40. Both layers' X are small which make unroll inefficient since it requires more global memory read than our baseline.*

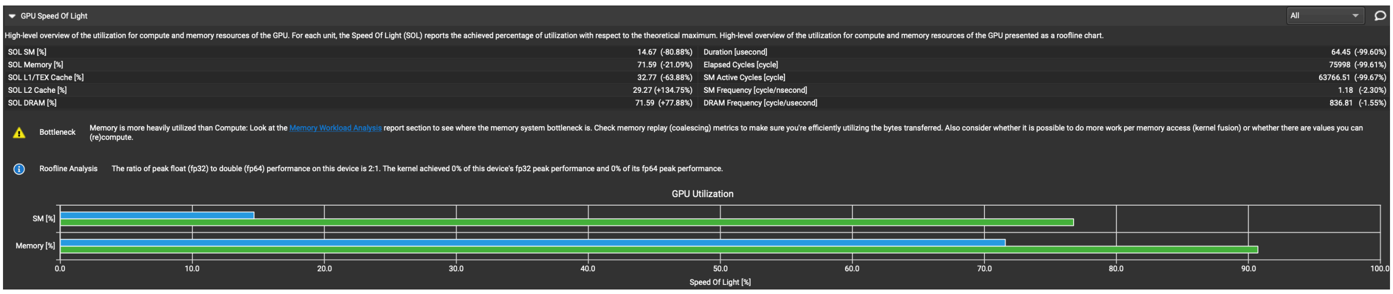
Compare between baseline and the optimization I implement, as we can see from the table above and data analysis below, the optimization makes the performance worse for all batch sizes. The OP time required for both layers increase obviously while the total execution time stays almost the same.

Particularly, we can obeserve from `nv-nsight-cu-cli` analysis that both `unroll_kernel` and `matrix_multiplication_kernel` have low utilization of GPU and Pipe compared to baseline, which make this optimization has longer OP time.

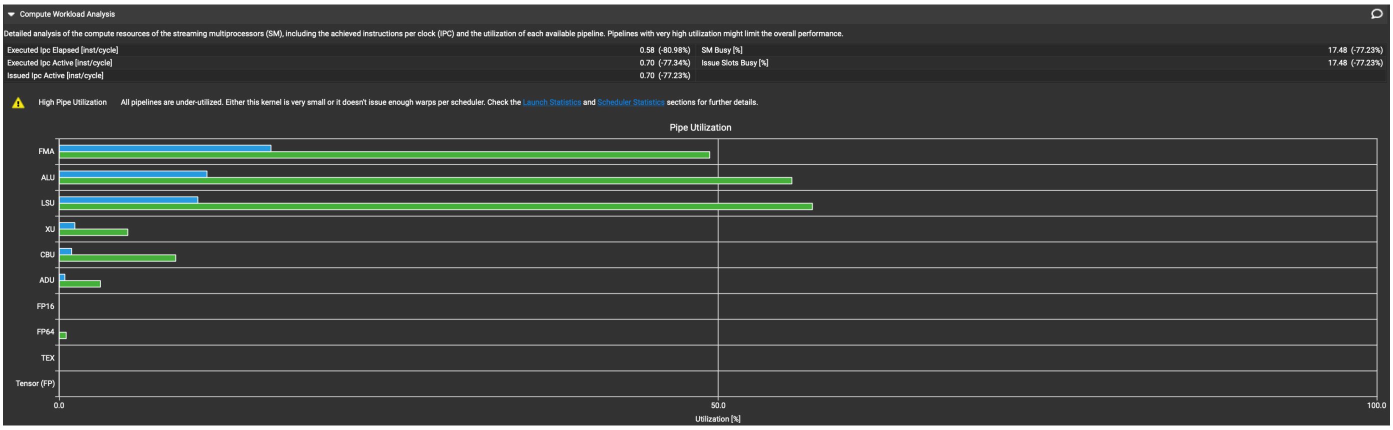
- ***Statistic analysis:***

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

Time(%)      Total Time       Calls          Average         Minimum         Maximum  Name
-------  --------------  ----------  ---------------  --------------  --------------  ------------------------------
   74.4      1033034966           8      129129370.8           19358       554758556  cudaMemcpy
   13.4       186462017          10       18646201.7           76358       182813588  cudaMalloc
   10.6       146842597          10       14684259.7           63781        75994382  cudaFree
    1.5        21116918        1604          13165.2            2785        15853923  cudaLaunchKernel
    0.0          138999           6          23166.5            2764          116402  cudaDeviceSynchronize




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average         Minimum         Maximum  Name
-------  --------------  ----------  ---------------  --------------  --------------  ------------------------------
   69.3       101344225         800         126680.3          107007          147807  matrixMultiply
   30.7        44898501         800          56123.1           47488           68576  unroll_kernel
    0.0            2976           2           1488.0            1408            1568  do_not_remove_this_kernel
    0.0            2720           2           1360.0            1344            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average         Minimum         Maximum  Name
-------  --------------  ----------  ---------------  --------------  --------------  ------------------------------
   91.3       938622997           2      469311498.5       384655756       553967241  [CUDA memcpy DtoH]
    8.7        89545191           6       14924198.5            1216        48008124  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations              Average            Minimum            Maximum  Name
-----------------  --------------  -------------------  -----------------  -----------------  ------------------------------
        1722500.0               2             861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6              89819.0              0.004           288906.0  [CUDA memcpy HtoD]
```

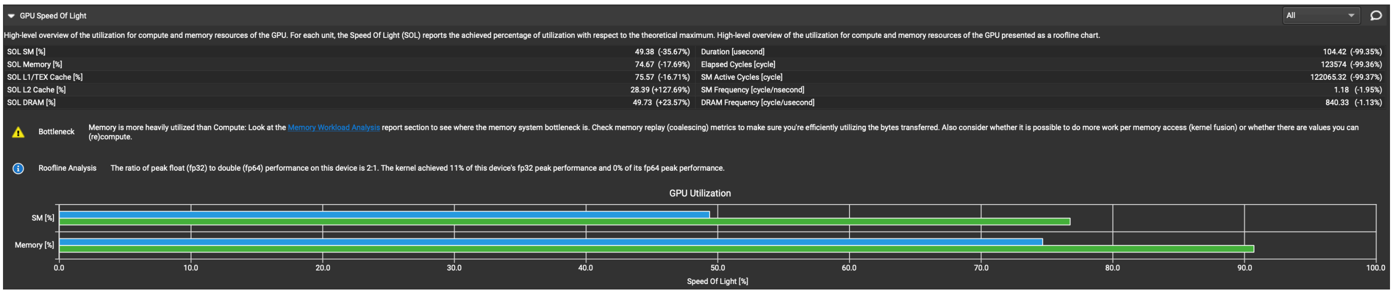
- ***unroll_kernel GPU analysis:***



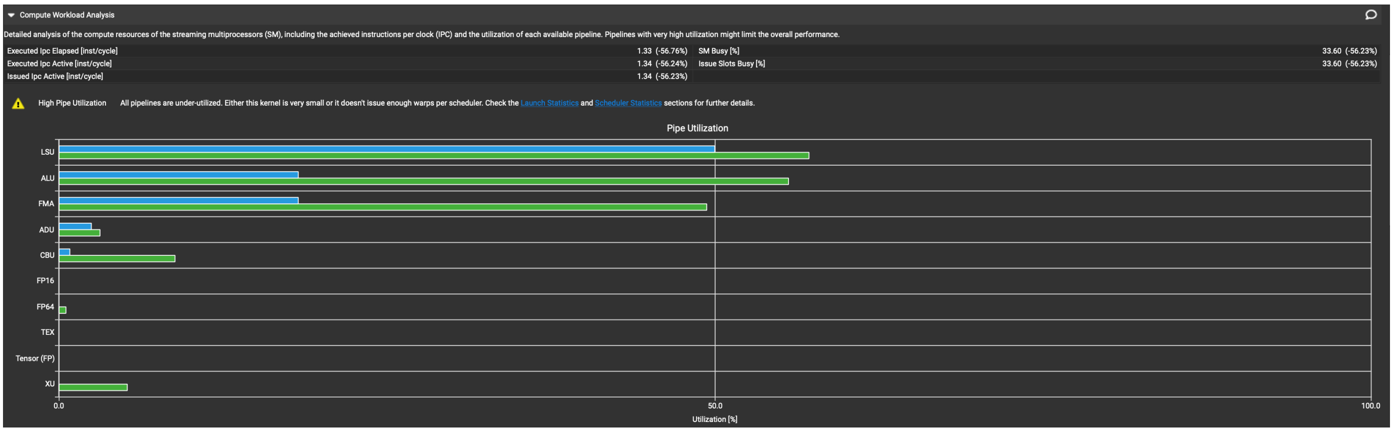
- ***unroll_kernel Pipe analysis:***



- ***matrix_multiplication_kernel GPU analysis:***



- ***matrix_multiplication_kernel Pipe analysis:***



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

I mainly refer to chapter 16 in the textbook.

Optimization 4

- `new-forward-v6.cu`
- Using Streams to overlap computation with data transfer
- Tiled shared memory convolution
- Multiple kernel implementation for different layer
- Sweeping various parameters to find best values

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

I choose streams to overlap computation as it could simultaneously execute a kernel while performing s copy between device and host memory. Multiple optimizations methods are applied in this optimizations since I am trying to get the best 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?

The overlap computation uses multiple streams to execute several kernels while perform data transfer at the same time. I think it could greatly improve performance as it allows concurrent copying and execution, which will reduce the total time needed.

Also, I check the size of different layer such that I can implement different kernels for different layer. For instance, I apply kernel call with TILE_WIDTH 16 for the first layer while I apply kernel call with TILE_WIDTH 8 for the second layer.

Moreover, I modify the parameters used in each kernels to find the best performance.

The optimization could synergize with other optimizations. I choose optimization 2 to build this optimization.

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	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*5.0432ms*	*4.323ms*	*9.88s*	*0.86*
1000	*56.494ms*	*50.4428ms*	*10.404s*	*0.886*
10000	*550.516ms*	*434.512ms*	*1m28.07s*	*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).

** <u>As this optimization focuses on optimizing the total time required, I will put the analysis of performance mainly on the total time instead of OP time.</u>*

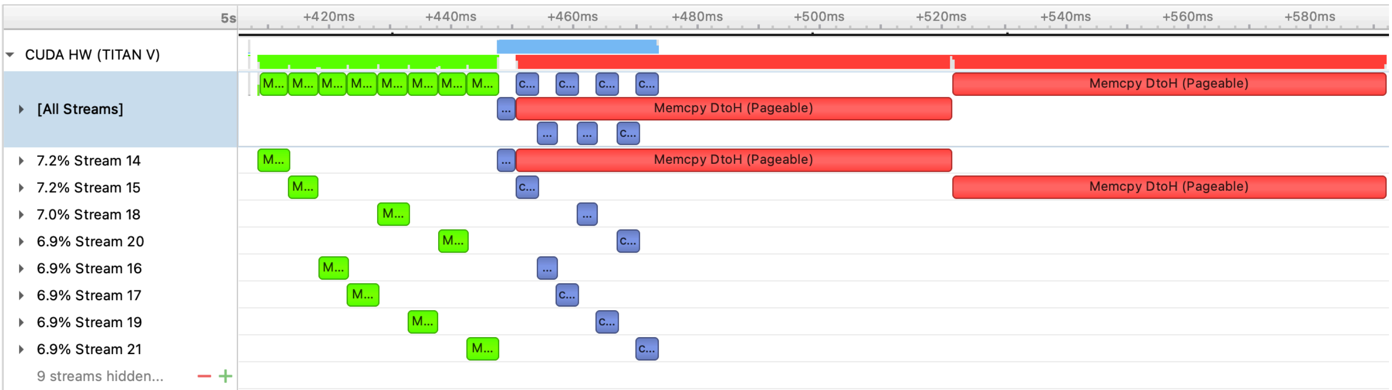
Yes, it improves the total performance as we consider the total layer time costed. Because by using streams, it allows concurrent copying and execution such that when convolution kernel is executed in a stream, the memory copy kernel will be executed in another stream at the same time. I creat 8 streams and hence 8 kernel calls will be handled simultaneously to improve the performance.

*As we can see from the timeline of overlap computation, where **green** parts represent **Memcpy() from Host to Device** and **blue** parts represent **kernel calls**, the kernel calls and Memcpy from Device to Host are overlapped for about 20ms. The optimization does have improved the total time needed but it is not obvious because Memcoy DtoH takes much more time than kernel calls.*

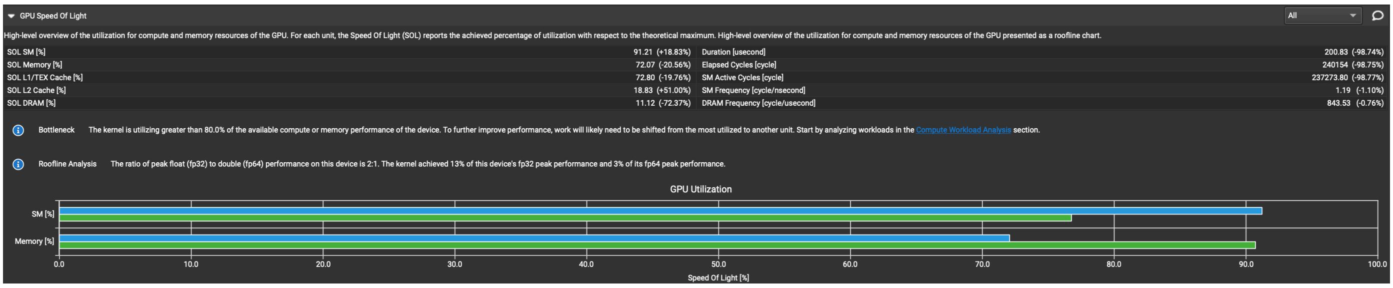
The analysis from `nsys profile` does not provide useful information and I will omit the analysis of it. The GPU and Pipe utilization should be similar to optimization 1 since I did not modify the kernel. This can be verified in the data sheet below, which is similar to optimization 1.

<u>According to the timeline of this optimizaiton, optimizing the data transfer from Device to Host should have a better improvement.</u>

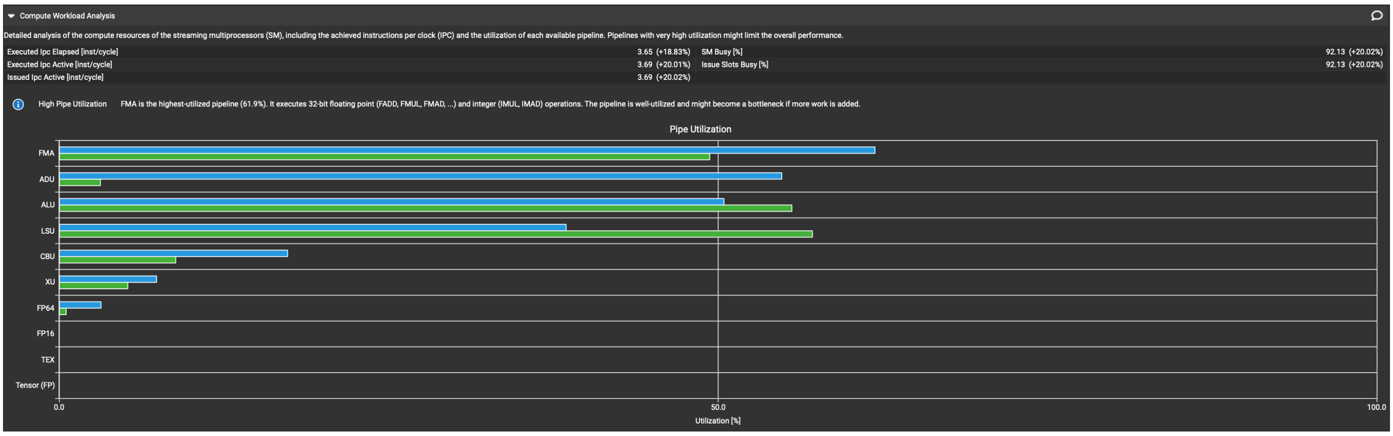
- ***Timeline Analysis of Overlap Optimization:***



- ***Shared memory convolution kernel GPU analysis:***



- ***Shared memory convolution kernel Pipe analysis:***



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

I mainly refer to the slides in the lecture.