ECE 408 Final Project Report

Team Name: Master of Webpu
Jack Danner (Netid: jackld2)
Kai Torng (Netid: tkai2)
Yihao Deng (Netid: ydeng29)
University of Illinois Urbana-Champaign

Milestone 4

Optimization 1: This optimization was done by exploiting the parallelism in batch size, the number of output feature maps, and the size of output images. The idea and code were referenced from chapter 16 of the readings given in this class[1]. From the CPU version of the convolution layer, we could see that the number of for loops can be reduced by associating the different loop parameters with thread blocks and threads by proper indexing. Furthermore, the calculations of different output feature maps in different batches of images are independent. Therefore, we identified this could be an optimization opportunity. As carrying out the calculations by each thread in a parallel manner, the overall throughput of our approach should drastically increase compared to the CPU version. We thought it would be fruitful since this degree of parallelism has reduced half of the outer for loops in the CPU implementation. And these loop parameters are B=10k, M=16, H_out=34, W_out=34. We utilized a grid with dimensions of batch size (x), number of output feature maps(y), and blocks per output image(z), along with 2-dimensional blocks with a size of TILE_WIDTH = 16. Each block is calculating a 16 by 16 portion of an output image map in a given batch. It was indeed fruitful as shown in figure 1. Compared to the baseline model in our milestone 3, the OpTime of this approach is about 3x less in the test of 10k images. Figure 3 demonstrates the dramatic increase (more than 3x in SM percentages compared to the baseline model) in the level of parallelism of our approach. This makes sense since we utilized 3x more threads to do the computation separately.

```
* Running bin/bash -c './m4 10000'   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Conv-GPU==

Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 16.184633 ms
Layer 1 OpTime: 16.217113 ms
Layer 1 LayerTime: 606.71443 ms
Layer 2 GPUTime: 61.390969 ms
Layer 2 OpTime: 61.424057 ms
Layer 2 LayerTime: 497.821795 ms
```

Figure 1: GPUTime, OpTime, and LayerTime of Optimization 1

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   83.9      1109068568           8     138633571.0           18065       563712164  cudaMemcpy
   15.2       200252989           8      25031623.6           74898       197311100  cudaMalloc
    0.6         8569174           6       1428195.7           21319         8447894  cudaLaunchKernel
    0.3         3596290           8        449536.2           72997         1846018  cudaFree
    0.0           17372           4          4343.0            2550            5151  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        79117459           2      39558729.5        16114510        63002949  conv_forward_kernel
    0.0            2880           2          1440.0            1440            1440  do_not_remove_this_kernel
    0.0            2720           2          1360.0            1312            1408  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   92.7       952220164           2     476110082.0       405458757       546761407  [CUDA memcpy DtoH]
    7.3        74926076           6      12487679.3            1184        40732881  [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]
```

Figure 2: Nsys Profiling

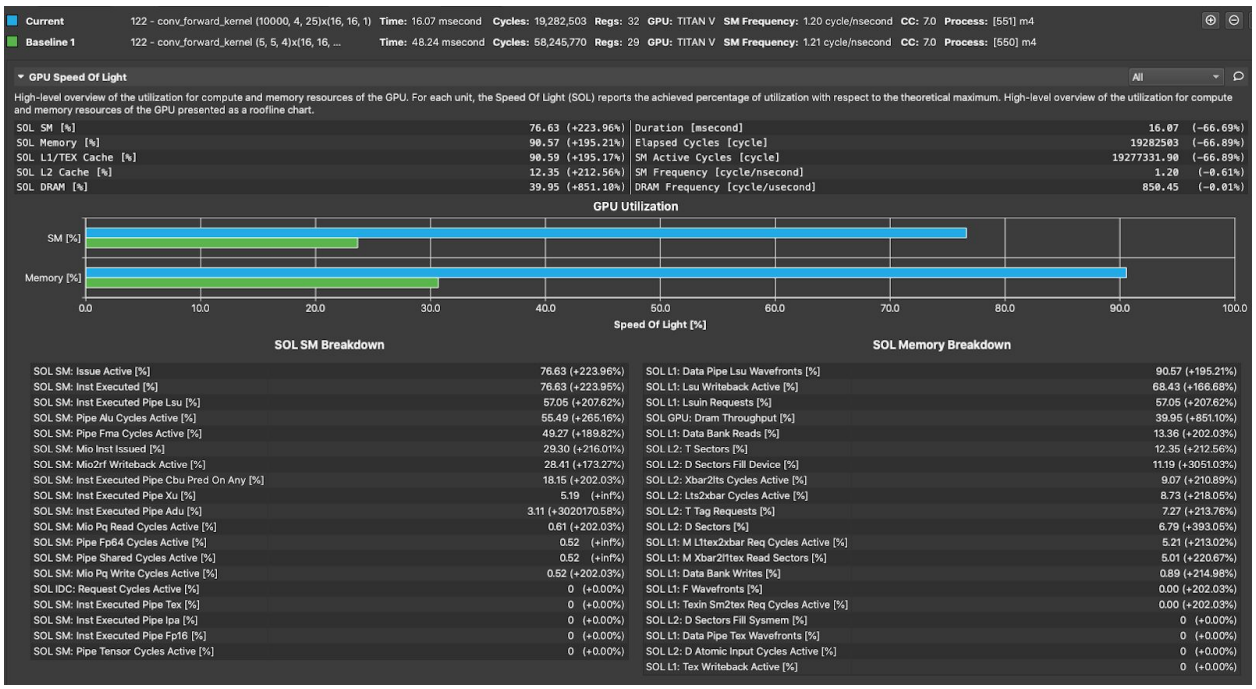


Figure 3: Comparison of GPU SOL utilization in Nsight-Compute GUI

Optimization 2: Our second optimization focuses on utilizing constant memory to store the convolutional kernels for the convolution process. In our baseline implementation as shown in Milestone 3 copies the entire host array of kernels to global memory on the device, and then accesses the kernel from global memory whenever a convolution is performed. Since the size of the host kernel array is relatively small (A maximum of 7*7*4*16 in the two layers we are testing), we thought it would be possible to load the entire kernel array into constant memory. With this, any kernel elements that are used multiple times will not need to be read from global memory multiple times, leading to a theoretical improvement in performance. Furthermore constant memory is great for entries that are being used by multiple threads at the same time.

```
Test batch size: 10000
-------------------------------
-          TIMINGS
-------------------------------
Layer 1 GPUTime: 47.98639 ms
Layer 1 OpTime: 48.00927 ms
Layer 1 LayerTime: 639.872587 ms
Layer 2 GPUTime: 167.378738 ms
Layer 2 OpTime: 167.409202 ms
```

```
Layer 2 LayerTime: 590.186315 ms
```

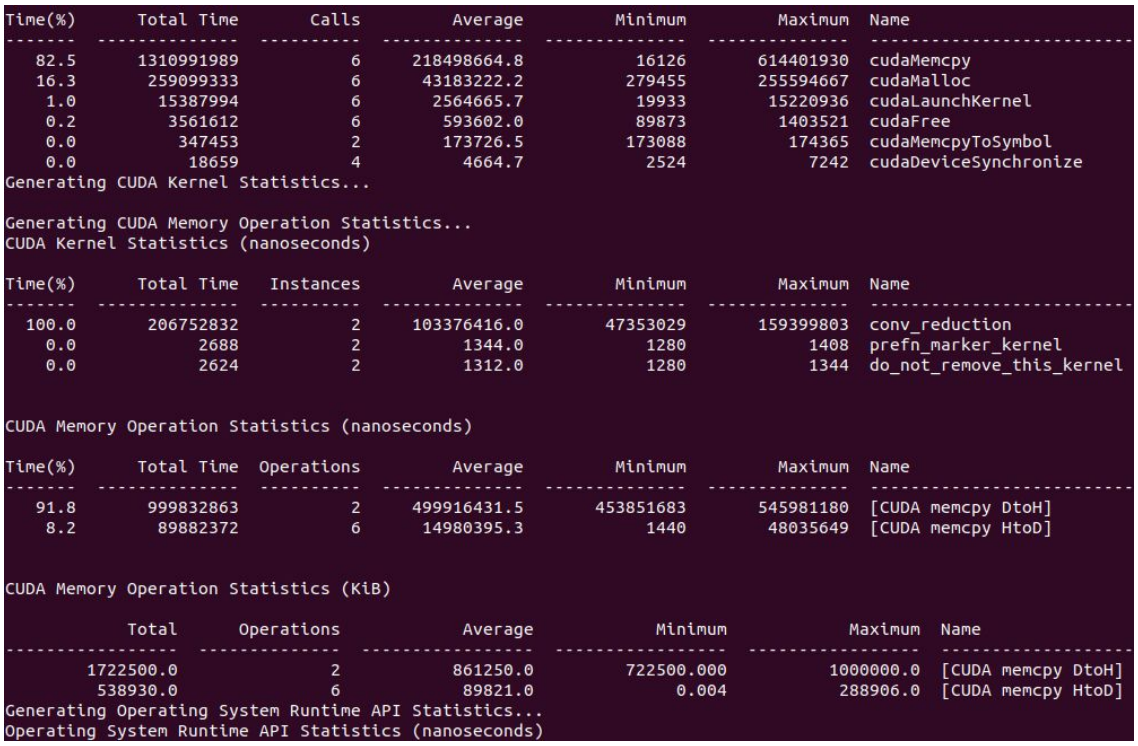
Figure 4:

Baseline kernel of M3 with batch size 10000.

```
Test batch size: 10000
--------------------------------
-          TIMINGS
--------------------------------
Layer 1 GPUTime: 47.743916 ms
Layer 1 OpTime: 47.759788 ms
Layer 1 LayerTime: 637.424793 ms
Layer 2 GPUTime: 161.691586 ms
Layer 2 OpTime: 161.713474 ms
Layer 2 LayerTime: 577.669299 ms
```

Figure 5:

Constant memory implementation on baseline kernel of M3 with batch size 10000. Shows a very slight OpTime improvement over the baseline.



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   82.5      1310991989           6     218498664.8        16126       614401930  cudaMemcpy
   16.3       259099333           6      43183222.2       279455       255594667  cudaMalloc
    1.0        15387994           6       2564665.7        19933        15220936  cudaLaunchKernel
    0.2         3561612           6        593602.0        89873         1403521  cudaFree
    0.0          347453           2        173726.5       173088          174365  cudaMemcpyToSymbol
    0.0           18659           4          4664.7         2524            7242  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
  100.0       206752832           2     103376416.0        47353029       159399803  conv_reduction
    0.0            2688           2          1344.0            1280            1408  prefn_marker_kernel
    0.0            2624           2          1312.0            1280            1344  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   91.8       999832863           2     499916431.5       453851683       545981180  [CUDA memcpy DtoH]
    8.2        89882372           6      14980395.3            1440        48035649  [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]
        538930.0               6            89821.0              0.004           288906.0  [CUDA memcpy HtoD]
Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)
```

Figure 6:

Nsys profile for constant memory implementation on baseline kernel of M3

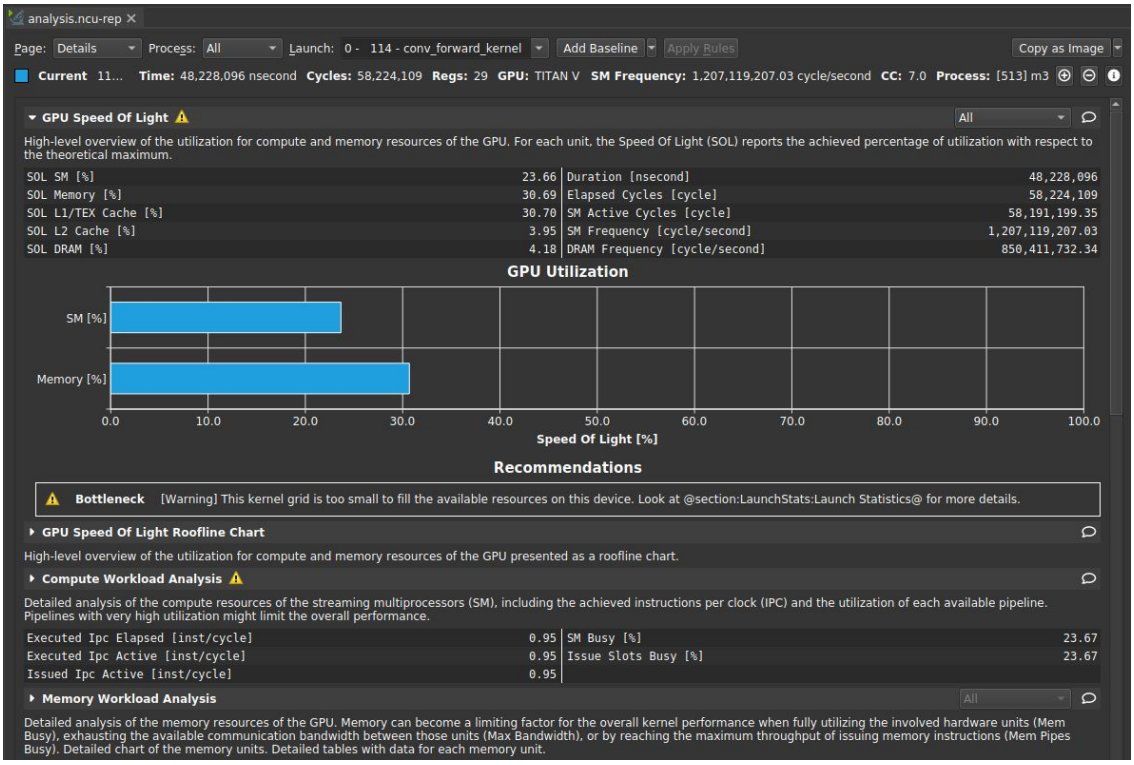


Figure 7:

Nv-nsight-cu-cli analysis on baseline kernel of M3

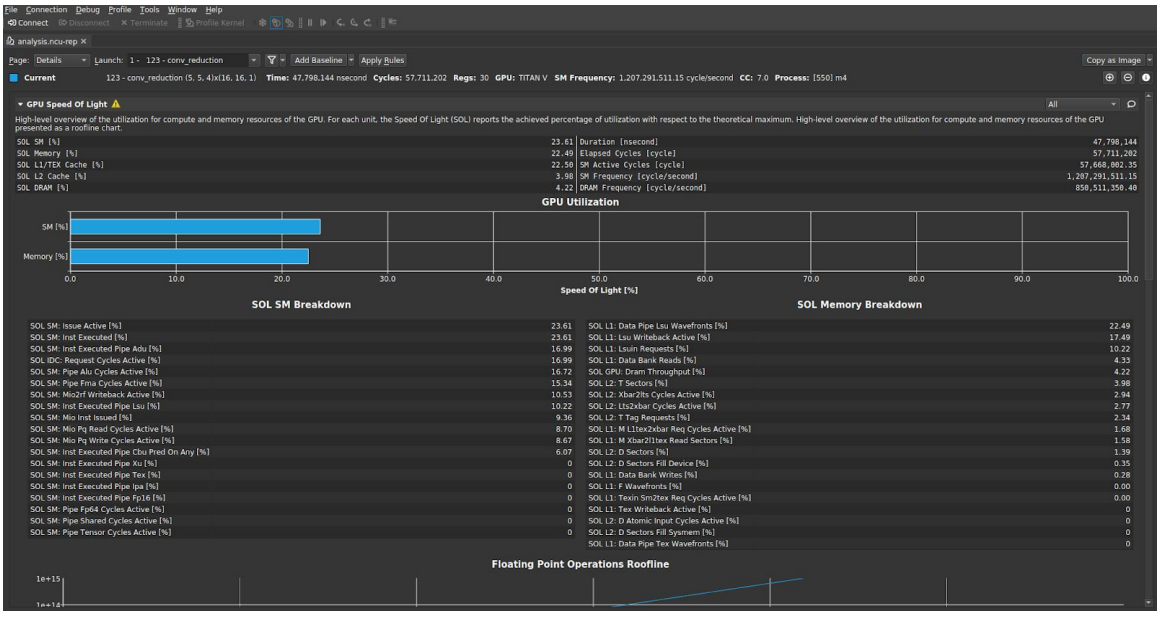


Figure 8:

Nv-nsight-cu-cli analysis of constant memory implementation on baseline kernel of M3

As you can see in the above figures, there is a negligible difference in OpTime between the baseline kernel and the constant memory assisted kernel. This surprised us because we thought that not having to read from global memory as much would lead to

a noticeable speedup. However it makes sense because the baseline we added it to is not highly parallel, so it makes sense that it didn't lead to much of a speedup because there is not a high amount of parallel threads reading from the constant memory.

Furthermore we are still reading from global memory for input array X in the same line, so we are reading from global memory anyway during that calculation, leading to a slowdown even though we have constant memory. However in figure 7 vs figure 8, we see that there is a quite noticeable reduction in memory bandwidth. This means if we use higher memory bandwidth in a more parallel kernel in the future, constant memory will be essential. Further shared memory optimizations in the future might benefit from the constant memory kernel storage more than the baseline milestone 3 kernel.

Optimization 3: Our third and final optimization focuses on performing tiled shared memory convolution. Our motivation is that we want to limit our reads from global memory. By using shared memory, most elements will only be read from global memory once in the beginning, and then calculations happen later in the kernel for each thread using shared memory. We thought that this would be a fruitful optimization because without global memory reads during calculation time, our kernel would perform in parallel much faster.

This was the most difficult task because we tried using the code from Chapter 16 [1] which does not quite work correctly for this application. We understand the pseudocode for chapter 16 where each block is responsible for calculating a tile portion of an output image, but has shared memory of size [TILE_WIDTH + K - 1][TILE_WIDTH + K - 1] and each block first loads all needed elements including halo elements into shared memory. However the code in the book does not handle indexing correctly and we were not able to pinpoint the bug even with the help of the TAs. We opted for a shared memory structure of size of [TILE_WIDTH]][TILE_WIDTH] where we only load main elements, and then halo elements are accessed from global memory, very similar to strategy 3 from lecture. Any threads that are out of bounds of our output image do nothing. We got some surprising results as shown below:

```
Test batch size: 10000
--------------------------------
-          TIMINGS
--------------------------------
Layer 1 GPUTime: 27.686454 ms
Layer 1 OpTime: 27.709622 ms
Layer 1 LayerTime: 644.06813 ms
Layer 2 GPUTime: 105.989589 ms
Layer 2 OpTime: 106.015061 ms
```

```
Layer 2 LayerTime: 564.234896 ms
```

Figure 9:

Parallelized kernel without shared memory. TILE_WIDTH = 16

```
Test batch size: 10000
--------------------------------
-          TIMINGS
--------------------------------
Layer 1 GPUTime: 38.507117 ms
Layer 1 OpTime: 38.527885 ms
Layer 1 LayerTime: 633.252256 ms
Layer 2 GPUTime: 146.917023 ms
Layer 2 OpTime: 146.966015 ms
Layer 2 LayerTime: 567.377846 ms
```

Figure 10:

Parallelized kernel with shared memory of TILE_WIDTHxTILE_WIDTH, halo elements are accessed from global memory. TILE_WIDTH = 16

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   79.2      1216970968           6     202828494.7           15579       586636366  cudaMemcpy
   19.6       300667722           6      50111287.0          334325       296425276  cudaMalloc
    1.1        16252382           6       2708730.3           16978        16135965  cudaLaunchKernel
    0.2         2573914           6        428985.7           78992          906522  cudaFree
    0.0          169403           2         84701.5           84636           84767  cudaMemcpyToSymbol
    0.0          142672           4         35668.0            2328          126325  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0       185507293           2      92753646.5        38271185       147236108  conv_forward_kernel
    0.0            4288           2          2144.0            1280            3008  do_not_remove_this_kernel
    0.0            3840           2          1920.0            1344            2496  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.7       951748181           2     475874090.5       404197453       547550728  [CUDA memcpy DtoH]
    7.3        74570223           6      12428370.5            1504        38830477  [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]
        538930.0               6           89821.0              0.004          288906.0  [CUDA memcpy HtoD]
```

Figure 11:

Nsys profile for shared memory implementation

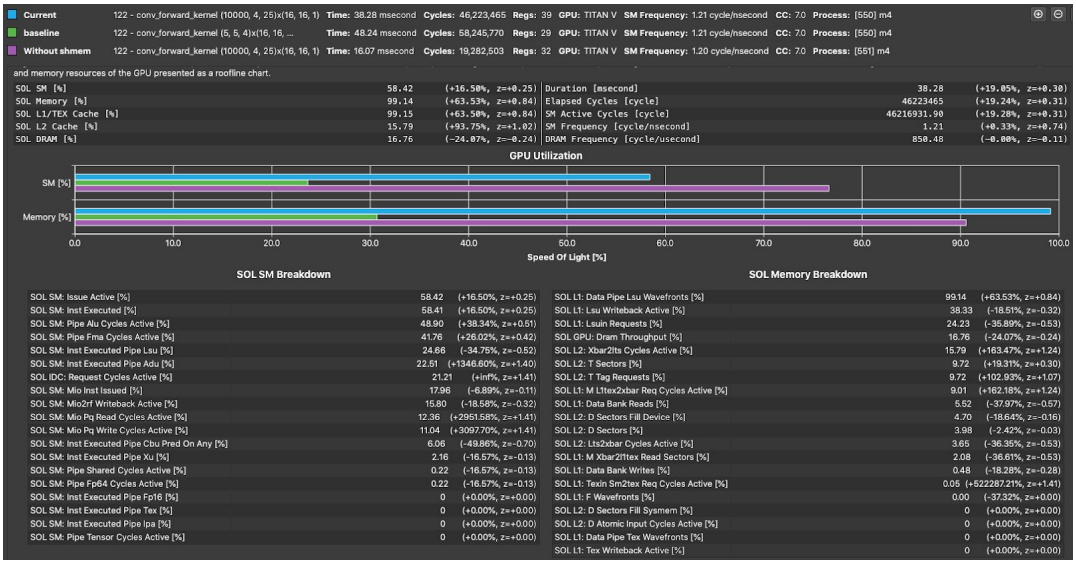


Figure 12:
Nv-nsight-cu-cli analysis of shared memory implementation

We discovered that performance on our shared memory version was actually worse than our non-shared memory version. The Nv-nsight analysis in Figure 12 shows that more memory bandwidth is being used during our tiled shared memory convolution compared to non-shared. This may be due to the fact that we are not reusing as many values as we thought. We are also still reading from global memory for the halo tiles, and also non-full tiles in the output handle shared memory in a slightly inefficient way because of how the bounds must be checked. Even though this is not the result we expected, it is likely that more shared memory considerations could be taken and tile width could be adjusted to improve performance. We will be exploring this further in the future, but for now this optimization with TILE_WIDTH x TILE_WIDTH shared memory did not yield better performance.

Contributions:

- Jack
    - Implemented Optimization 2 constant memory
    - Wrote report on optimization 2 and 3
    - Assisted with getting chapter 16 code to work for milestones 1 and 3,
    - Implemented a different form of a tiled shared memory than book, went to office hours to fix bugs on shared memory code.
- Yihao
    - Implemented optimization 1: exploiting parallelism of the batch size, number of output feature maps, and the image size of output feature.
    - Gathered analysis data and wrote report on optimization 1

- Provided the structure and the block layout of the kernel for shared memory optimization based on chapter 16 from the readings

References:

[1] Ginsburg, B. 3rd-Edition-Chapter16-case-study-DNN-FINAL-corrected.

Milestone 2

```
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-CPU==
Op Time: 102186 ms
Conv-CPU==
Op Time: 355462 ms
Test Accuracy: 0.8714
real 9m17.605s
user 9m16.176s
sys 0m1.212s
```

Milestone 3

Output Rai running GPU implementation

```
Test batch size: 100
Loading fashion-mnist data...Done
```

```
Loading model...Done
Conv-GPU==
Op Time: 93.3388 ms
Conv-GPU==
Op Time: 8.89906 ms
Test Accuracy: 0.86
real  0m1.127s
user  0m0.992s
sys   0m0.124s
```

```
Test batch size: 1000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Op Time: 224.33 ms
Conv-GPU==
Op Time: 59.5902 ms
Test Accuracy: 0.886
real  0m9.836s
user  0m9.502s
sys   0m0.288s
```

```
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Op Time: 818.089 ms
Conv-GPU==
Op Time: 613.797 ms
Test Accuracy: 0.8714
real  1m37.090s
user  1m35.506s
sys   0m1.612s
```

Demonstrate nsys profiling the GPU execution:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   80.6      1236614178           6     206102363.0          157481       582106622  cudaMemcpy
   19.0       292000049           6      48666674.8           75563       289535677  cudaMalloc
    0.4         5419913           6        903318.8           68079         3322905  cudaFree
    0.0          250315           2        125157.5           25507          224808  cudaLaunchKernel
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0       213850804           2     106925402.0        48242514       165608290  conv_forward_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   91.2       924323560           2     462161780.0       391208278       533115282  [CUDA memcpy DtoH]
    8.8        89613222           4      22403305.5            1184        48025139  [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               4           134729.0              0.766           288906.0  [CUDA memcpy HtoD]
Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)
```

List of all kernels that collectively consume more than 90% of the program time:

Since there is only one kernel in this case, the conv_forward_kernel would be the only one consuming more than 90% of the program time

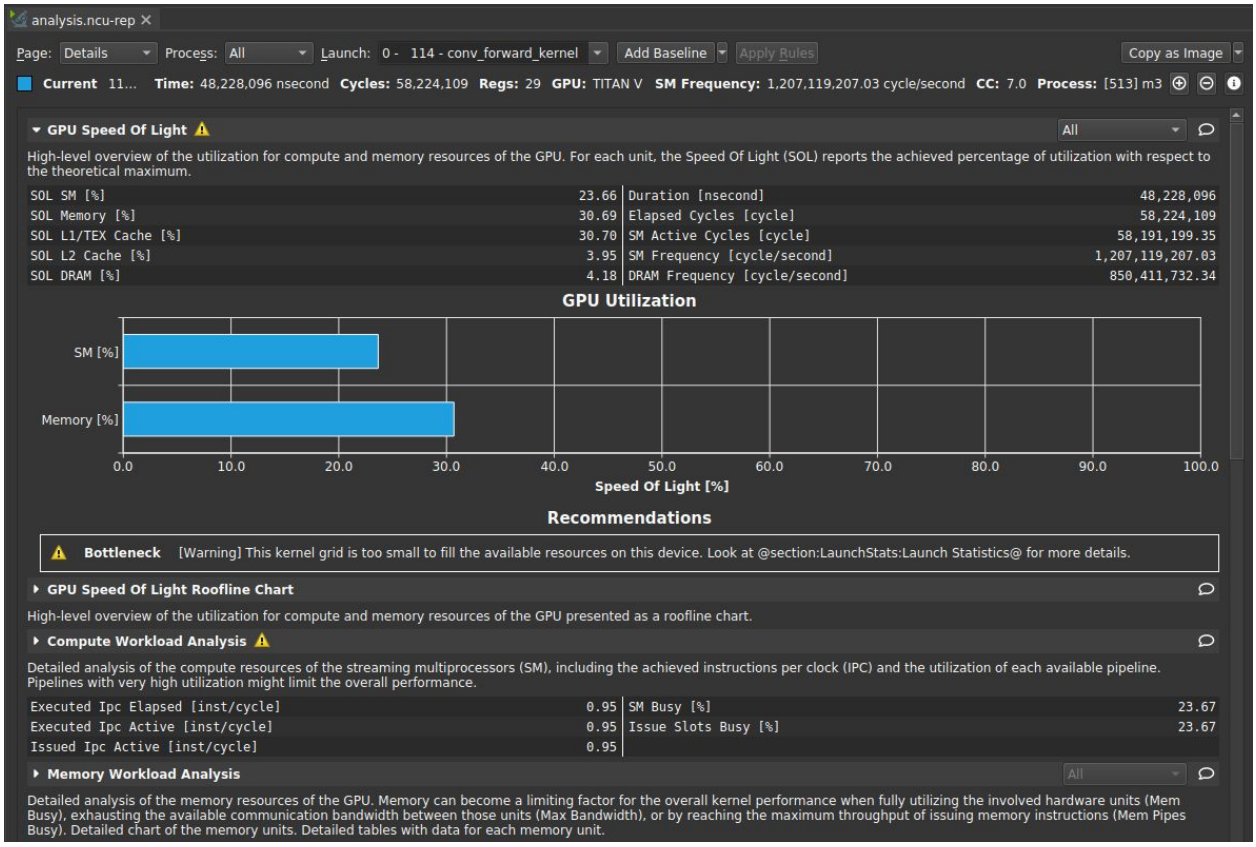
List of all CUDA API calls that collectively consume more than 90% of the program time:

cudaMemcpy takes up 80.6% of the program time, and cudaMalloc takes up 19.0%, so both cudaMemcpy and cudaMalloc.

Difference between kernels and API calls:

A kernel is a function that is able to be executed multiple times on the gpu. The kernel must be written with this in mind. It has access to the thread block and thread index of the thread that the kernel is being executed on. API calls are functions that are executed on the host system to assist and prepare execution, such as allocate and deallocate memory on the GPU. An API call is also used to set block and grid dimensions for example.

Screenshot of GPU SOL utilization in Nsight-Compute GUI:


analysis.ncu-rep
Page: Details
Process: All
Launch: 0 - 114 - conv_forward_kernel
Add Baseline
Apply Rules
Copy as Image
Current 11...
Time: 48,228,096 nsecond
Cycles: 58,224,109
Regs: 29
GPU: TITAN V
SM Frequency: 1,207,119,207.03 cycle/second
CC: 7.0
Process: [513] m3
GPU Speed Of Light
All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%] 23.66
SOL Memory [%] 30.69
SOL L1/TEX Cache [%] 30.70
SOL L2 Cache [%] 3.95
SOL DRAM [%] 4.18
Duration [nsecond] 48,228,096
Elapsed Cycles [cycle] 58,224,109
SM Active Cycles [cycle] 58,191,199.35
SM Frequency [cycle/second] 1,207,119,207.03
DRAM Frequency [cycle/second] 850,411,732.34
GPU Utilization
SM [%]
Memory [%]
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0
Speed Of Light [%]
Recommendations
Bottleneck [Warning] This kernel grid is too small to fill the available resources on this device. Look at @section:LaunchStats:Launch Statistics@ for more details.
GPU Speed Of Light Roofline Chart
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle] 0.95
Executed Ipc Active [inst/cycle] 0.95
Issued Ipc Active [inst/cycle] 0.95
SM Busy [%] 23.67
Issue Slots Busy [%] 23.67
Memory Workload Analysis
All
Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.