REPORT

Final Submission: unrolling and matrix-multiplication

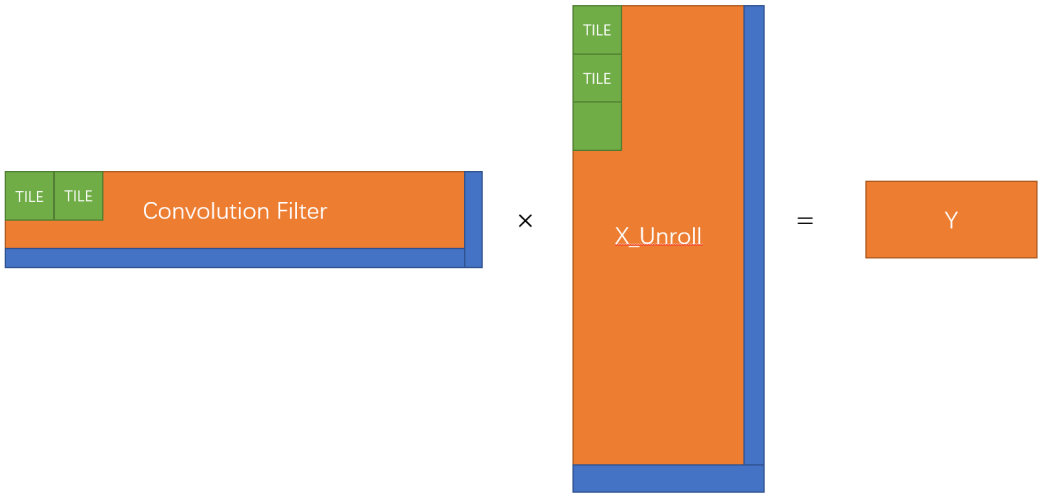
Junda Feng

jundaf2@illinois.edu

Table of Contents

Final Submission: unrolling and matrix-multiplication

In the final submission, we are going to implement the forward convolution layers using a different approach: shared memory matrix multiplication and input matrix unrolling. The key idea in this checkpoint is illustrated in the schematic diagram below.



Baseline

For the baseline implementation of the Matrix-Multiplication convolution forward kernel with unrolling, we choose two use to kernels to achieve the requirements in the slides. One kernel for unrolling the input feature maps in batch and one kernel for tiled matrix-matrix multiplication. When executing the convolutional layer of a batch size 10000, the following memory error happens.

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Unroll Success!
num_W_Unroll_Rows x num_W_Unroll_Columns : 4 x 49
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Shared MatMul Success!
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
Layer Time: 85.7982 ms
Op Time: 37.7239 ms
Conv-GPU==
CUDA error: an illegal memory access was encountered
```

However, when the batch sizes are 5000, 1000 and 100 as shown in the following picutres, there is no such accuracy error and the network reaches desired accuracy.

```
* Running bash -c "./final 5000"   \\ Output will appear after run is complete.
Test batch size: 5000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Unroll Success!
num_W_Unroll_Rows x num_W_Unroll_Columns : 4 x 49
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Shared MatMul Success!
B,M,C,H,W,K is : 5000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
Layer Time: 362.121 ms
Op Time: 56.7042 ms
Conv-GPU==
Unroll Success!
num_W_Unroll_Rows x num_W_Unroll_Columns : 16 x 196
num_X_Unroll_Rows x num_X_Unroll_Columns : 196 x 1156
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Shared MatMul Success!
B,M,C,H,W,K is : 5000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 343.893 ms
Op Time: 93.4633 ms

Test Accuracy: 0.871
```

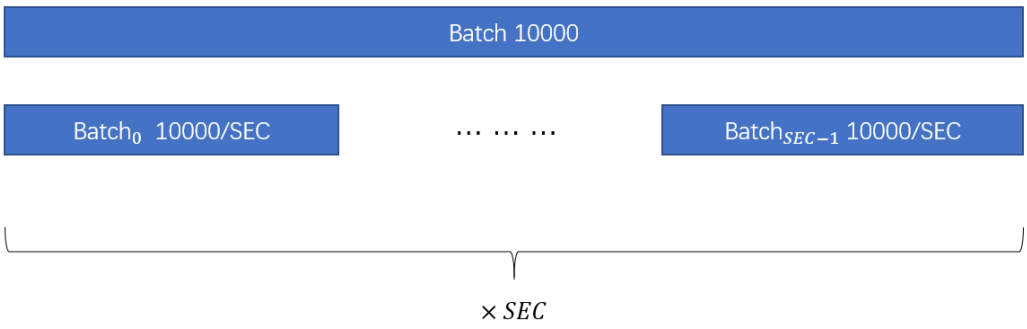
```
* Running bash -c "./final 1000"   \\ Output will appear after run is complete.
Test batch size: 1000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_W_Unroll_Rows x num_W_Unroll_Columns : 4 x 49
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
B,M,C,H,W,K is : 1000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 71.4616 ms
Op Time: 11.5622 ms
Conv-GPU==
num_W_Unroll_Rows x num_W_Unroll_Columns : 16 x 196
num_X_Unroll_Rows x num_X_Unroll_Columns : 196 x 1156
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
B,M,C,H,W,K is : 1000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,2,2,4
Layer Time: 61.3267 ms
Op Time: 16.6518 ms

Test Accuracy: 0.886
```

```
* Running bash -c "./final 100"   \\ Output will appear after run is complete.
Test batch size: 100
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_W_Unroll_Rows x num_W_Unroll_Columns : 4 x 49
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
B,M,C,H,W,K is : 100,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 8.17185 ms
Op Time: 1.3769 ms
Conv-GPU==
num_W_Unroll_Rows x num_W_Unroll_Columns : 16 x 196
num_X_Unroll_Rows x num_X_Unroll_Columns : 196 x 1156
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
B,M,C,H,W,K is : 100,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,2,2,4
Layer Time: 7.43869 ms
Op Time: 1.87692 ms

Test Accuracy: 0.86
```

Based on the output of the linux CLI together with the code analysis, we think the error is due to the limitation of device memory when the batch size is too large to fit into physical storage of the device. After some calculation, we find that the memory requirement of the unrolled input with a batch size of 10000 is

$$3.3 \times \text{sizeof(float)}\ \text{GBytes}$$

which is indeed too large. Thus we decide to divide the total batch into smaller equal-sized batches and lauch the kernels (one for unroll the input and another for shared matrix multiplication) for SEC times as illustrated in the following figure.



One thing also need to be mentioned is when using cudaMalloc() to allocate memory for a smaller batch of size B/SEC, the expression should be

```
1. cudaMalloc((void**)(&device_X_unroll),
   (C*K*K*h_out*w_out*(B/SEC))*sizeof(float));
```

rather than

```
1. cudaMalloc((void**)(&device_X_unroll),
   (C*K*K*h_out*w_out*B/SEC)*sizeof(float));
```

because the maximum integer that can be expressed in C++ language is 4294967295 even use unsigned long int, which is smaller than (C*K*K*h_out*w_out*B) when B=10000. Thus we need to enclose B/SEC using parentheses due to the fact that multiplication is evaluated first in left-to-right order in C/C++. The result with correct accuracy and a tile width of 32 is shown in the figure below, which demonstrates the correctness of our baseline implementation.

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
memory allocated for one section: 6272000000 bytes
num_W_Unroll_Rows x num_W_Unroll_Columns : 4 x 49
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
Layer Time: 640.395 ms
Op Time: 81.829 ms
Conv-GPU==
memory allocated for one section: 4531520000 bytes
num_W_Unroll_Rows x num_W_Unroll_Columns : 16 x 196
num_X_Unroll_Rows x num_X_Unroll_Columns : 196 x 1156
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 490.459 ms
Op Time: 81.0322 ms

Test Accuracy: 0.8714
```

The nsys CLI output is as follows.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
   67.6       976358999           8     122044874.9           19584       522926399  cudaMemcpy
   20.2       291238804          10      29123880.4           71109       275649622  cudaMalloc
   10.9       156974068          14      11212433.4             870        22000662  cudaDeviceSynchronize
    1.2        16938066          12       1411505.5            4272        16776763  cudaLaunchKernel
    0.2         2773492           8        346686.5           60290          840670  cudaFree
    0.0          127494           2         63747.0           10771          116723  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
   52.6        82511621           4      20627905.2        19073067        21996953  unroll_kernel
   47.4        74410100           4      18602525.0        17160982        20046692  conv_forward_matmul_kernel
    0.0            2656           2          1328.0            1280            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
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
   92.7       895642450           2     447821225.0       373546790       522095660  [CUDA memcpy DtoH]
    7.3        70589773           8       8823721.6            1472        37662936  [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]
        538932.0               8            67366.0              0.004           288906.0  [CUDA memcpy HtoD]
```

As we can see, the input feature maps with unrolling (DtoH) will use much more memory compared with the non-unrolled raw input (HtoD).

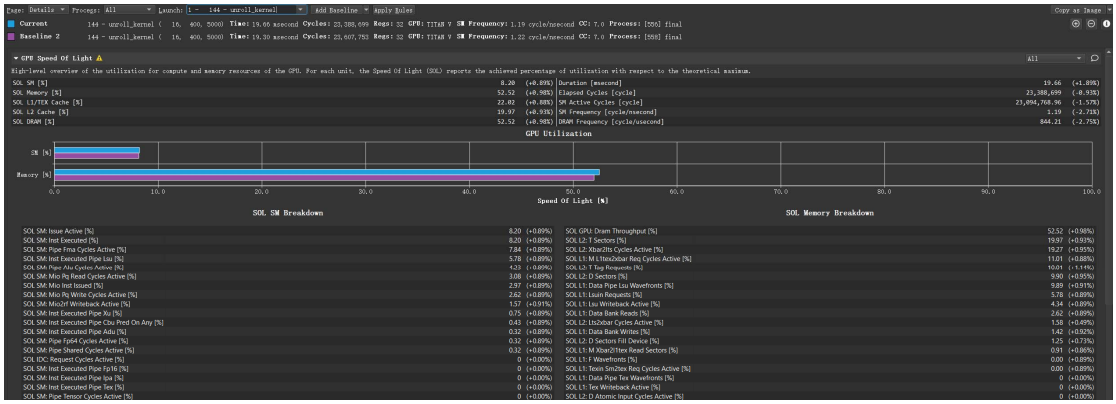
Constant memory for convolution masks.

To utilize constant memory in convolution, we put all kernels in constant memory instead of fetching them from global memory and loading them into shared memory each time. To reduce branch divergence, instead of directly using the constant memory kernels used in milestone 3, we choose to pad the convolutional filters first according to the tile width used in matrix-matrix multiplication and then move them to the constant memory through cudaMemcpyToSymbol(). As we can see from the CLI output with a tile width of 32 in the below figure, the unrolled convolutional filter weight matrix has a different size compared with the non-padded ones without using constant memory.

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
memory allocated for one section: 6272000000 bytes
num_W_Unroll_Rows x num_W_Unroll_Columns : 32 x 64
num_X_Unroll_Rows x num_X_Unroll_Columns : 49 x 6400
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
Layer Time: 641.296 ms
Op Time: 85.154 ms
Conv-GPU==
memory allocated for one section: 4531520000 bytes
num_W_Unroll_Rows x num_W_Unroll_Columns : 32 x 224
num_X_Unroll_Rows x num_X_Unroll_Columns : 196 x 1156
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 517.25 ms
Op Time: 105.447 ms

Test Accuracy: 0.8714
```

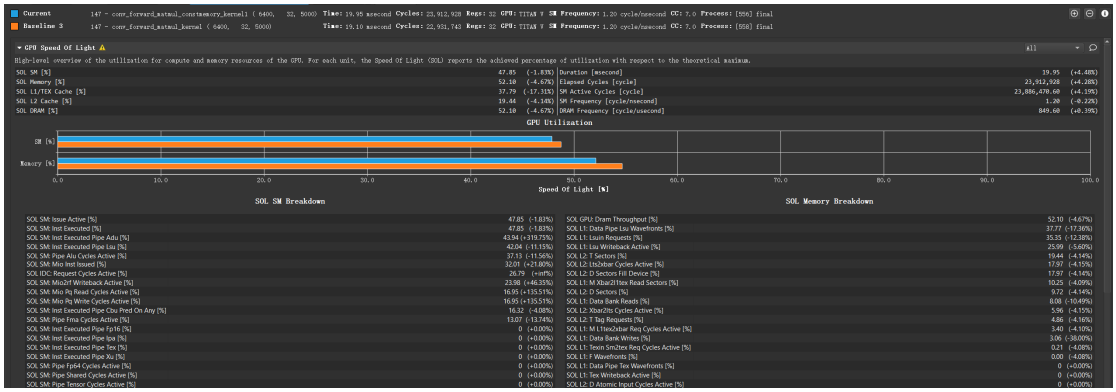
This is actually a little bit slower than the baseline using shared memory for both matrices in matrix multiplication. To solve this puzzle of reduced efficiency, we choose to analyze the cause by interpreting nv-nsight-cu output which is visualized using Nsight Compute GUI.

- unroll kernel comparison

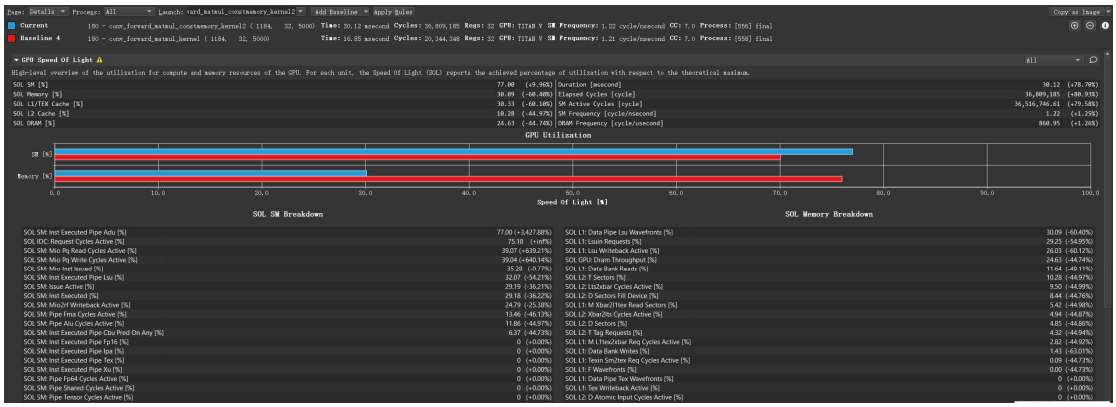


As we can see, the unroll kernel performance is almost the same for filter weight in shared memory (purple) and filter weight in constant memory (blue). Thus we can say the efficiency reduction is cost by the matrix multiplication kernel.

- mat-mul kernel comparison



As we can see, for the first layer, the mat-mul kernel with filter weight in constant memory (blue) takes slightly longer time than the mat-mul kernel with filter weight in shared memory (orange). From the figure, we can tell that this may due to the lower utilization of the SM.



But for the second layer, the mat-mul kernel with filter weight in constant memory (blue) takes significantly longer time than the mat-mul kernel with filter weight in shared memory (red) even though the two are using the same gridDim and blockDim. My current guess for this phenomemon based on the metrics in the Nsight Compute GUI is that the device can not pipelining the data loading from constant memory to register, which results in limited optimization in the data fetching.

Matrix-Multiplication with built-in unrolling.

The unrolling method is inherently a kind of mapping between the X_Unroll and the original input array. Because we have already explicitly calculated the locations of matrix elements of both of the matrices in shared memory multiplication, we can do inverse engineering to calculate the element location in the original input array very easily.

Also, the CUDA kernel statistics in the previous two implementations in which the unroll kernel and the forward matrix-multiplication kernel are separate suggest that the unroll kernel takes almost half of the time and has to be launched for multiple times due to the large batch size. We think the built-in unrolling can overlap some of the time and reduce the total amount of time the layer would take.

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 613.062 ms
Op Time: 50.6304 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 479.318 ms
Op Time: 63.9665 ms

Test Accuracy: 0.8714
```

Loop through the tile width of matrix-matrix multiplication kernel

Because the best kernel is the one with built-in unrolling so far, we choose to loop through the width of the square tile of the matrix-matrix multiplication kernel with built-in unrolling and count the time consumption of layer execution to tell which tile width is the best. The tile width we choose to test ranges from 4 to 32 with a step size of 4.

- Tile width 4

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 670.811 ms
Op Time: 90.4534 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 652.624 ms
Op Time: 234.102 ms

Test Accuracy: 0.8714
```

- Tile width 8

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 614.084 ms
Op Time: 50.6184 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 475.125 ms
Op Time: 61.9061 ms

Test Accuracy: 0.8714
```

- Tile width 12

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 611.932 ms
Op Time: 54.2743 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 486.833 ms
Op Time: 75.7252 ms

Test Accuracy: 0.8714
```

- Tile width 16

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 596.531 ms
Op Time: 50.1144 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 438.365 ms
Op Time: 38.2574 ms

Test Accuracy: 0.8714
```

- Tile width 20

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 608.914 ms
Op Time: 56.8156 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 443.816 ms
Op Time: 43.0464 ms

Test Accuracy: 0.8714
```

- Tile width 24

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 616.345 ms
Op Time: 60.707 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 459.182 ms
Op Time: 44.7116 ms

Test Accuracy: 0.8714
```

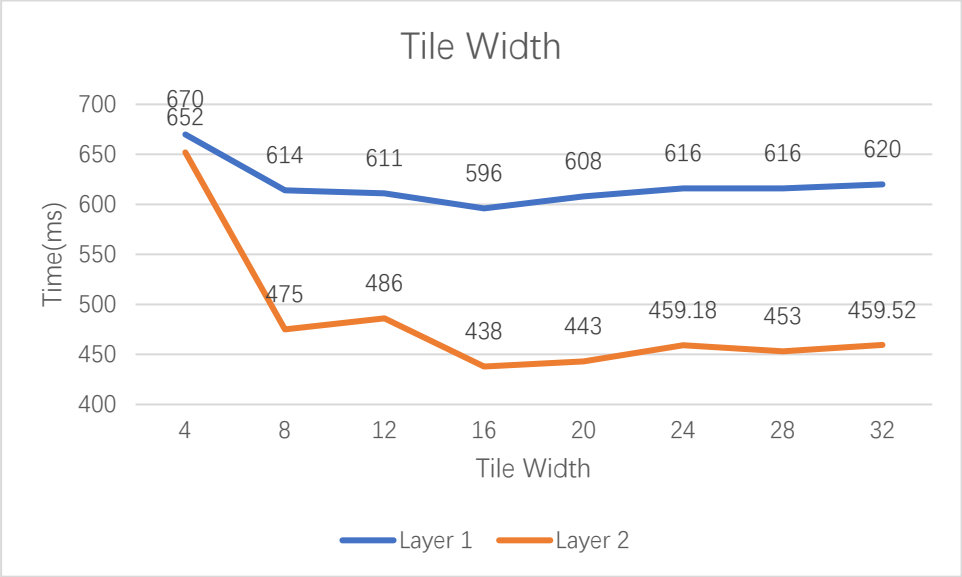
- Tile width 28

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 616.741 ms
Op Time: 64.5598 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 453.076 ms
Op Time: 48.9588 ms

Test Accuracy: 0.8714
```

- Tile width 32

```
* Running bash -c "./final"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 4 x 6400
Layer Time: 620.259 ms
Op Time: 61.5805 ms
Conv-GPU==
num_Y_Unroll_Rows x num_Y_Unroll_Columns : 16 x 1156
Layer Time: 459.524 ms
Op Time: 50.9944 ms

Test Accuracy: 0.8714
```

If we plot the layer time as follows, we can easily find that the best tile width is 16, which cost the least amount of time for both layers.



Milestone 3 – commonly used optimizations

Introduction

In this Milestone, we implemented five basic optimizations that are commonly used in GPU CUDA programming. They are shared memory convolution, weight matrix in constant memory, tuning with restrict and loop unrolling, multiple kernel implementations for different layer sizes and fixed point (FP16) arithmetic. In addition, we slightly swept block sizes to find the optimal value for performance improvement. In each optimization, we compared the performance with the baseline to show its impact. We also show the screen shots of the RAI output and analyze the profiler reports when justifying the modifications to the baseline in this report.

In reality, some of the optimizations may or may not necessarily give better performance. For example, based on the content of our lecture, using constant memory suppose to give a better performance but on modern GPUs, caching is done automatically through the L1 cache. Because we are introducing some new bottlenecks and inefficiencies by implementing the optimizations, it is a common thing that our optimizations do not give better performance.

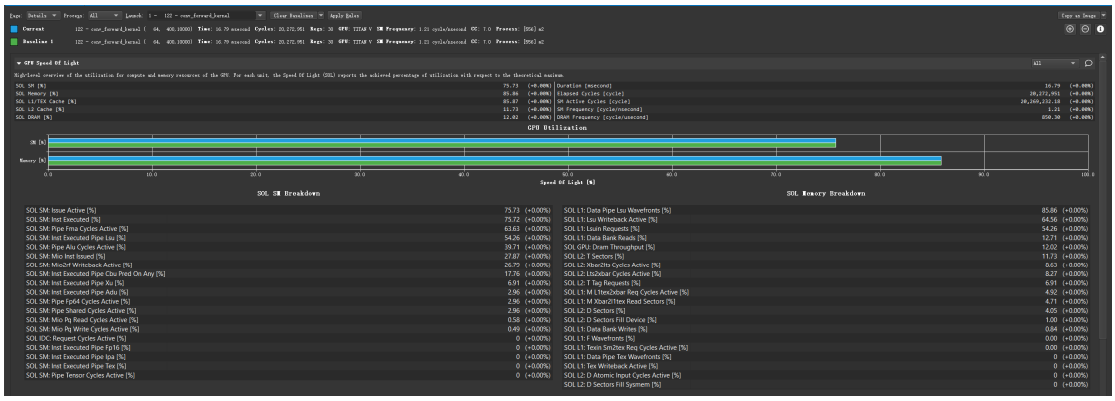
Baseline

This one is almost the same as the one in MT2. Other optimizations will be compared with this baseline to demonstrate their individual impact on the performance.

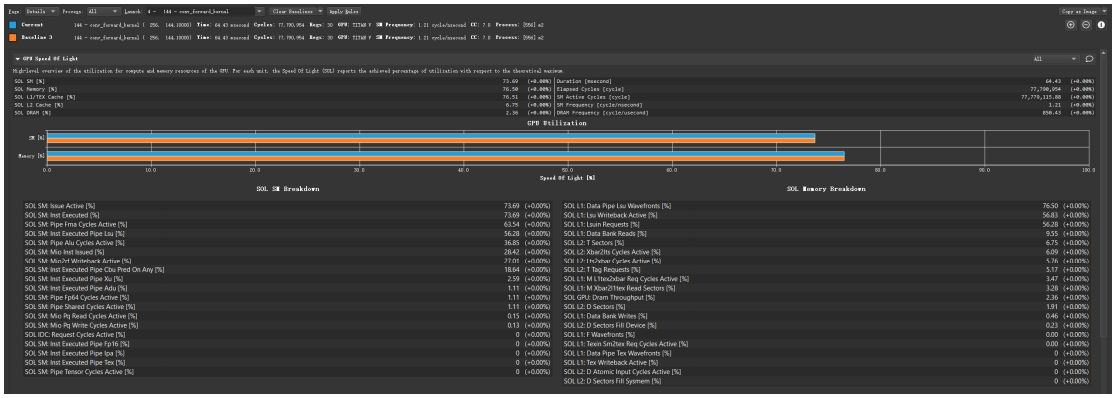
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,-1811583872,32764,16
Layer Time: 726.569 ms
Op Time: 25.9557 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,-1811583872,32764,4
Layer Time: 501.932 ms
Op Time: 74.4937 ms

Test Accuracy: 0.8714
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

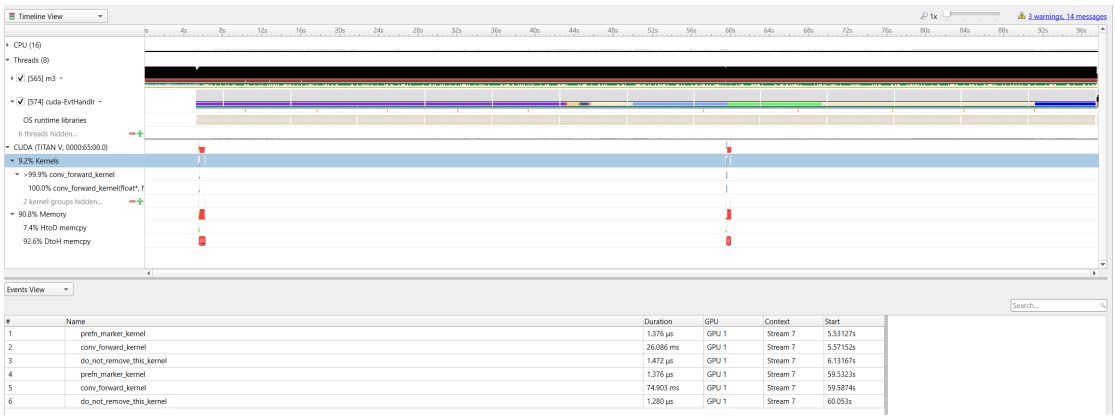


- second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   79.0      1030419054           8     128802381.8           11684       554224270  cudaMemcpy
   13.9       181330463           8      22666307.9           71186       177966066  cudaMalloc
    5.6        73306573           8       9163321.6            1255        55456733  cudaDeviceSynchronize
    1.2        16264769           6       2710794.8           16023        16151120  cudaLaunchKernel
    0.2         2512515           8        314064.4           63614          777100  cudaFree
    0.0           18311           2          9155.5            7467           10844  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        73279971           2      36639985.5        17824693        55455278  conv_forward_kernel
    0.0            2784           2          1392.0            1280            1504  do_not_remove_this_kernel
    0.0            2752           2          1376.0            1376            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.5       948186397           2     474093198.5       394799852       553386545  [CUDA memcpy DtoH]
    7.5        76402410           8       9550301.3            1472        40475875  [CUDA memcpy HtoD]
```



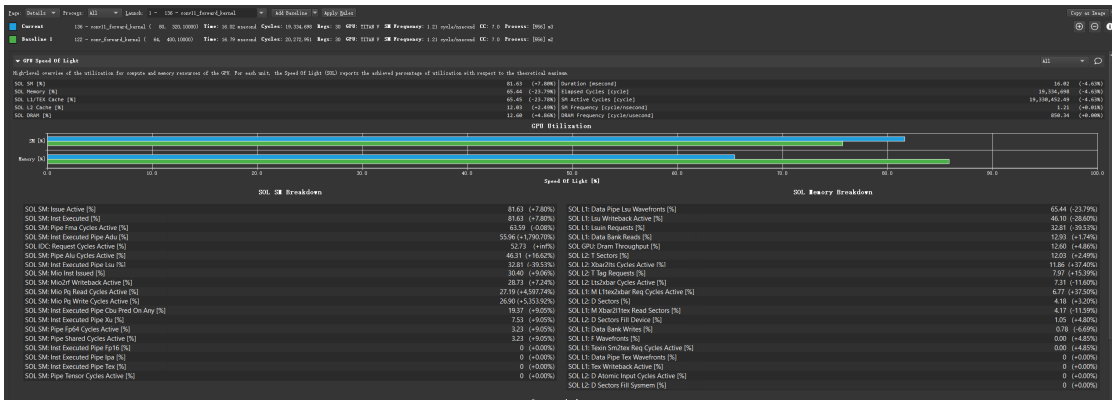
Optimization 1: Weight matrix (kernel values) in constant memory

When using this optimization, the kernel no longer need to fetch the filter data from the global memory. Thus, as can be seen from the Nsight Compute GUI, the time consumption is indeed smaller than the baseline.

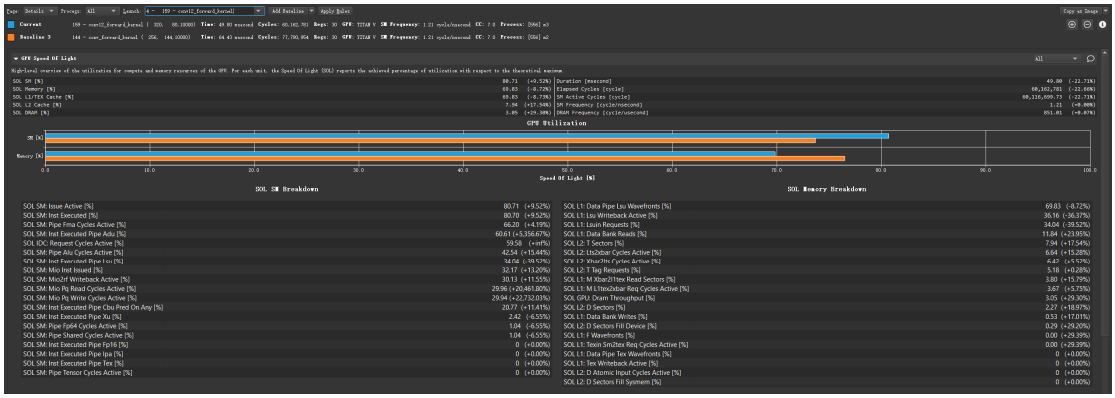
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 660.949 ms
Op Time: 23.7352 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 490.792 ms
Op Time: 64.1551 ms

Test Accuracy: 0.8714
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

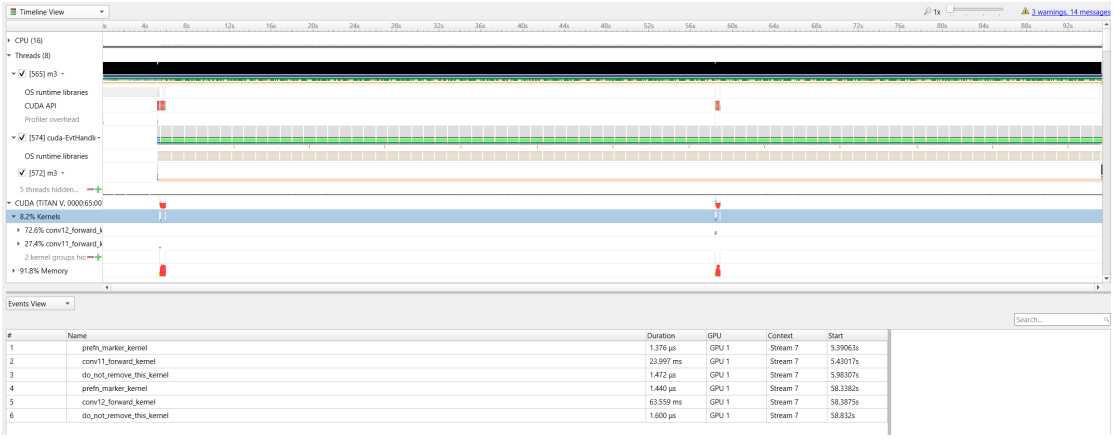


- second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   78.8       995184402           8     124398050.3           17925       527657219  cudaMemcpy
   12.9       163052591           8      20381573.9           77348       159690719  cudaMalloc
    6.9        87582050           8      10947756.3            1126        63560014  cudaDeviceSynchronize
    1.1        14141723           6       2356953.8           14630        14028137  cudaLaunchKernel
    0.2         2539370           8        317421.3           61657          764941  cudaFree
    0.0           18316           2          9158.0            7785           10531  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   72.6        63559096           1      63559096.0        63559096        63559096  conv12_forward_kernel
   27.4        23997416           1      23997416.0        23997416        23997416  conv11_forward_kernel
    0.0            3072           2          1536.0            1472            1600  do_not_remove_this_kernel
    0.0            2816           2          1408.0            1376            1440  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.7       905946928           2     452973464.0       379108768       526838160  [CUDA memcpy DtoH]
    7.3        71779545           8       8972443.1            1504        38183099  [CUDA memcpy HtoD]
```

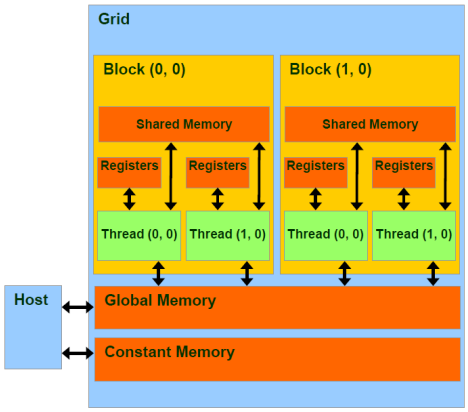


Optimization 2: 2D Shared Memory convolution && Optimization 1

Shared memory, which is shared by all threads in a thread block, has lower latency and about 10 times more bandwidth than global device memory. To take advantage of shared memory convolution, input data are loaded to shared memory prior to computation. Since the DRAM that is for global memory is slower than the SRAM that is for shared memory. This optimization can reduce memory bandwidth by letting threads in a thread block use the smaller but faster local shared memory. By first loading the required data into the shared memory once, we are able to reuse it for multiple times without accessing global memory. This optimization can improve the data reuse and decease the memory access time and thus improves the performance.

Programmer View of CUDA Memories

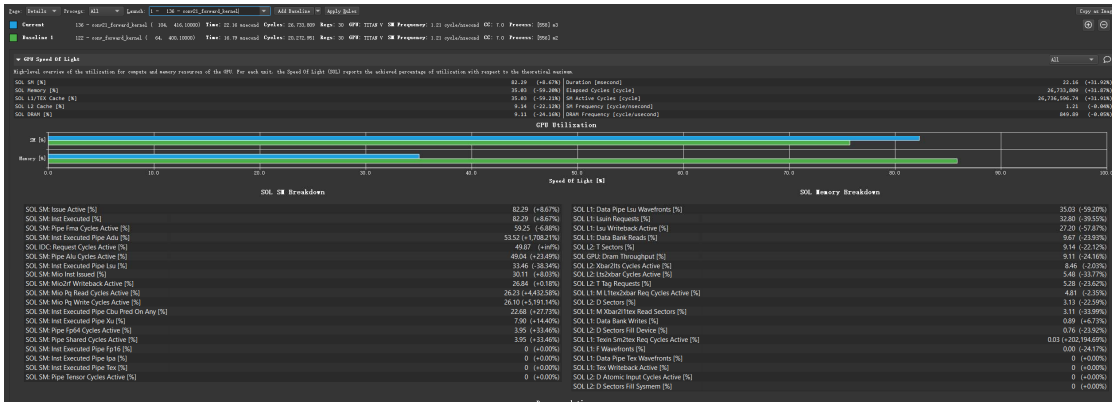
- Each thread can:
  - read/write per-thread **registers (~1 cycle)**
  - read/write per-block **shared memory (~5 cycles)**
  - read/write per-grid **global memory (~500 cycles)**
  - read/only per-grid **constant memory (~5 cycles with caching)**



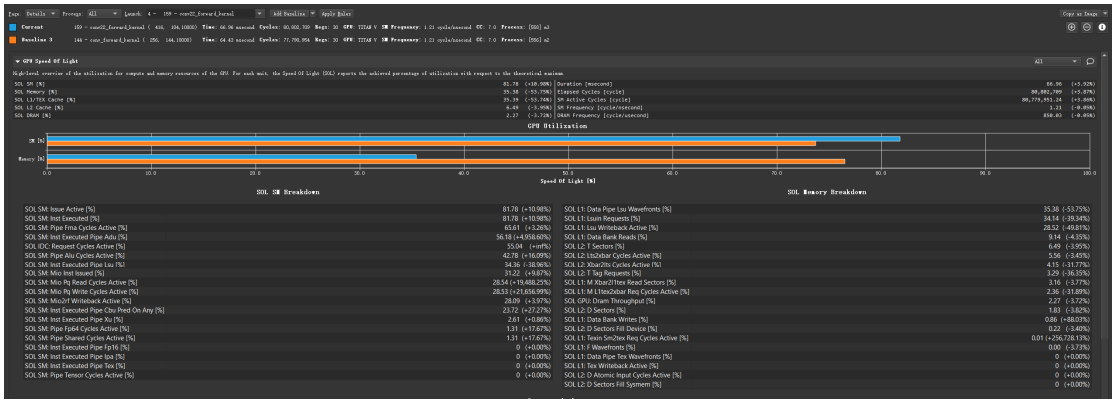
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 613.397 ms
Op Time: 27.1949 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,2,2,4
Layer Time: 508.137 ms
Op Time: 78.359 ms

Test Accuracy: 0.8714
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

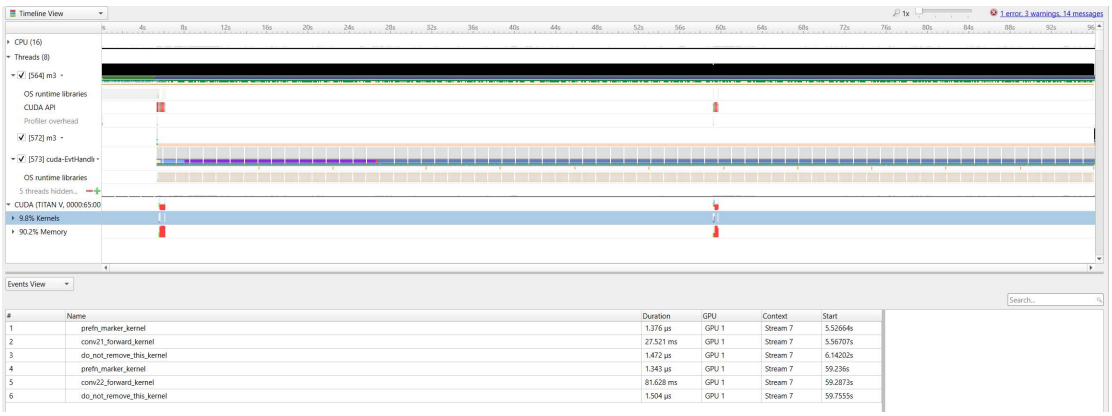


  - second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   77.3      1008308500           8     126038562.5           17677       546013879  cudaMemcpy
   12.9       168419438           8      21052429.7           70239       164945632  cudaMalloc
    8.4       109171442           8      13646430.3            1068        81628940  cudaDeviceSynchronize
    1.2        15531326           6       2588554.3           23928        15384372  cudaLaunchKernel
    0.2         2658003           8        332250.4           67132          819626  cudaFree
    0.0           18052           2          9026.0            7623           10429  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   74.8        81627991           1      81627991.0        81627991        81627991  conv22_forward_kernel
   25.2        27521360           1      27521360.0        27521360        27521360  conv21_forward_kernel
    0.0            2976           2          1488.0            1472            1504  do_not_remove_this_kernel
    0.0            2719           2          1359.5            1343            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   92.7       929878068           2     464939034.0       384734983       545143085  [CUDA memcpy DtoH]
    7.3        73234167           8       9154270.9            1504        39027481  [CUDA memcpy HtoD]
```



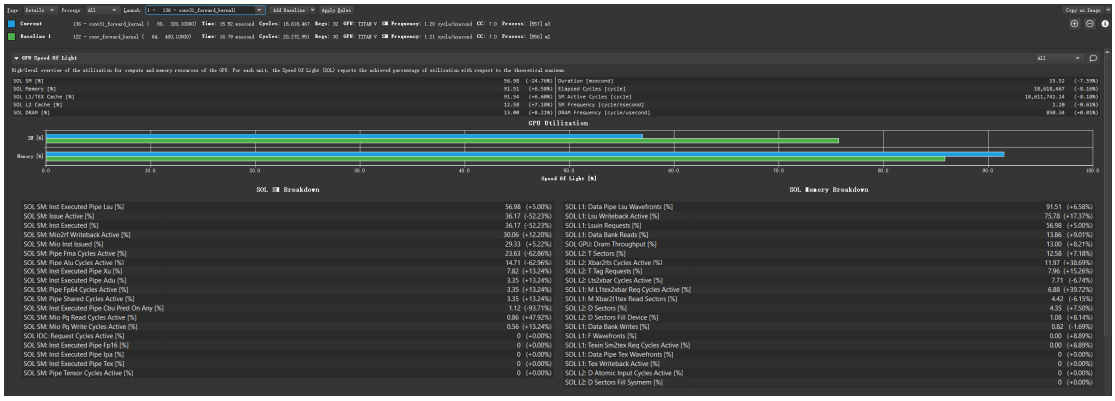
Optimization 3: Tuning with restrict and loop unrolling

The use of restricted pointer can let the compiler know the arrays have no overlap and thus reduce the number of instructions. By loop unrolling, the compiler can improve the instruction scheduling and minimize the cost of loop overhead. We do this by completely expanding the three inner loops related to the convolution. For the first layer, there is only one input feature map and for the second layer, there are four input feature maps. Thus the number of lines of code of loop unrolling part of the second layer is four times longer than that of the first layer. So based on the result shown in the Nsight Compute GUI, we can say that the optimization using loop unrolling can improve the kernel performance at the cost of increasing the difficulty of writing code.

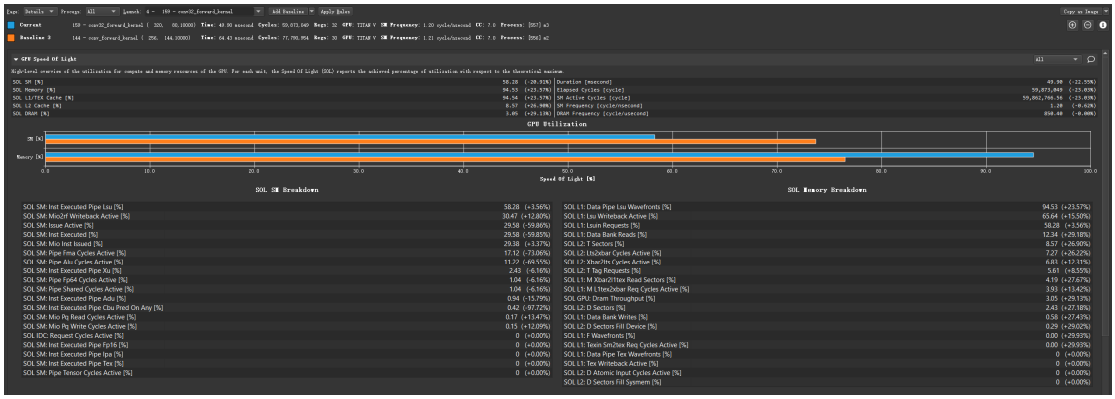
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 613.405 ms
Op Time: 20.9577 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 500.092 ms
Op Time: 55.8167 ms

Test Accuracy: 0.8714
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

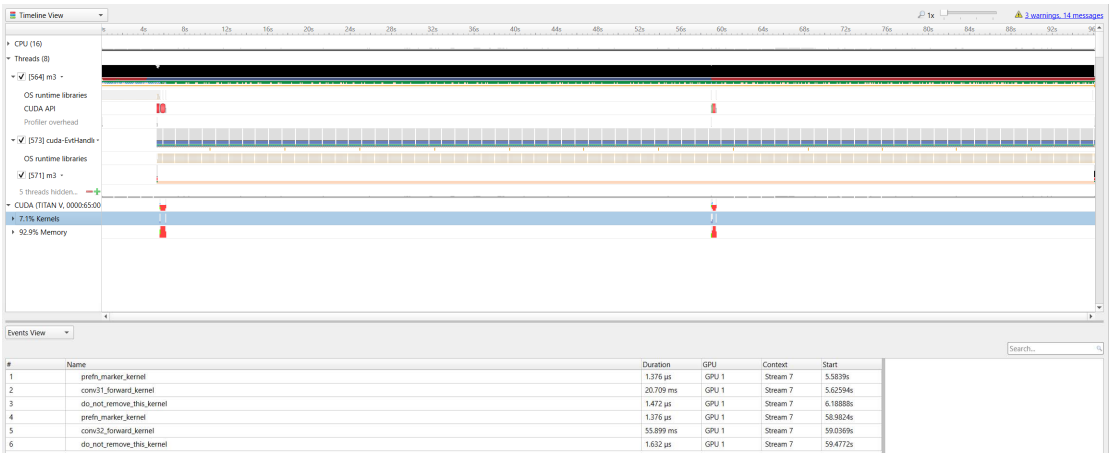


  - second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   75.4      1005391872           8     125673984.0           16956       540943185  cudaMemcpy
   17.5       232896302           8      29112037.7           71198       228327344  cudaMalloc
    5.8        76633335           8       9579166.9            1020        55899534  cudaDeviceSynchronize
    1.1        15098154           6       2516359.0           15896        14973795  cudaLaunchKernel
    0.2         2669989           8        333748.6           62527          796851  cudaFree
    0.0           25446           2         12723.0           11738           13708  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   73.0        55899085           1      55899085.0        55899085        55899085  conv32_forward_kernel
   27.0        20708706           1      20708706.0        20708706        20708706  conv31_forward_kernel
    0.0            3104           2          1552.0            1472            1632  do_not_remove_this_kernel
    0.0            2752           2          1376.0            1376            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   92.3       922727238           2     461363619.0       382605488       540121750  [CUDA memcpy DtoH]
    7.7        77193804           8       9649225.5            1472        40653546  [CUDA memcpy HtoD]
```



Optimization 4: Fixed point (FP16) arithmetic

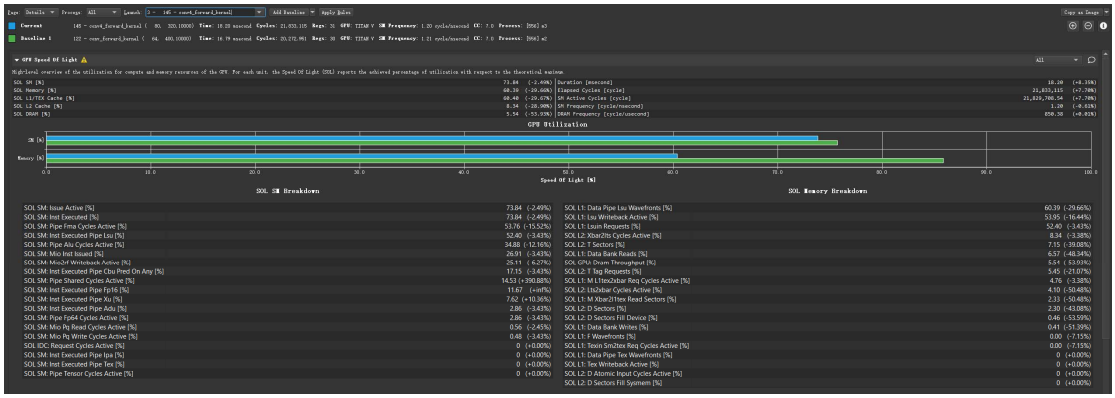
In computing, half precision (sometimes called FP16) is a binary floating-point computer number format that occupies 16 bits (two bytes in modern computers) in computer memory. They can express values in the range ±65,504, with the minimum value above 1 being 1 + 1/1024. (WikiMili, 2021) To be more specific, we can use one kernel to convert the data in the default format of floating point number to fixed point numbers to reduce the amount of data that needs to move from device memory to the GPU and do half-precision computations using specific APIs provided by CUDA. Once the convolution computation is finished, we can converted it back using another kernel. As we can see from the SOL GPU utilization, the memory required in this Fixed point arithmetic kernel is significantly less than that of the baseline implementation. As for the time consumption, the time required by the this optimization (also taken into account the addition kernels)

that do the convolution operations is indeed shorter than that of the baseline at the cost of implementing more kernels for data type conversion.

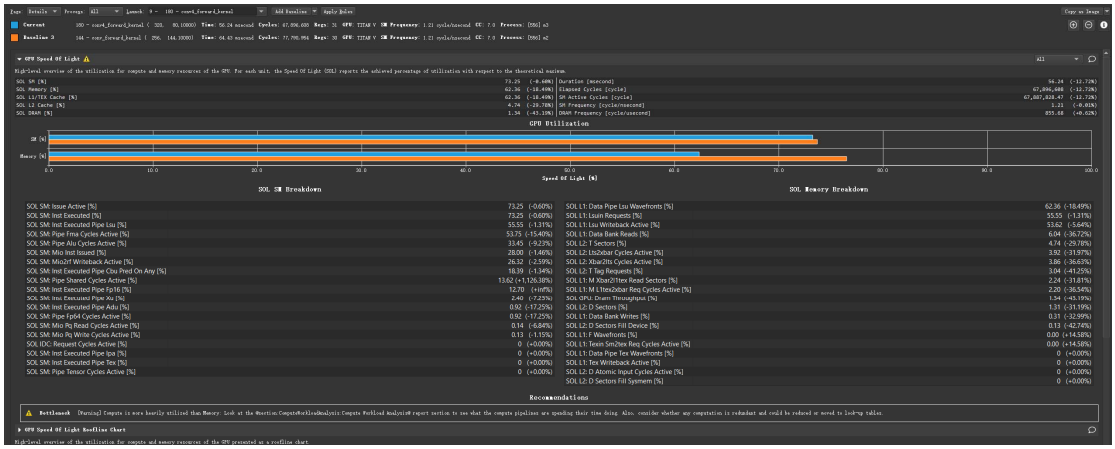
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 632.687 ms
Op Time: 21.5414 ms
Conv-GPU==
Layer Time: 497.994 ms
Op Time: 68.3053 ms

Test Accuracy: 0.8716
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

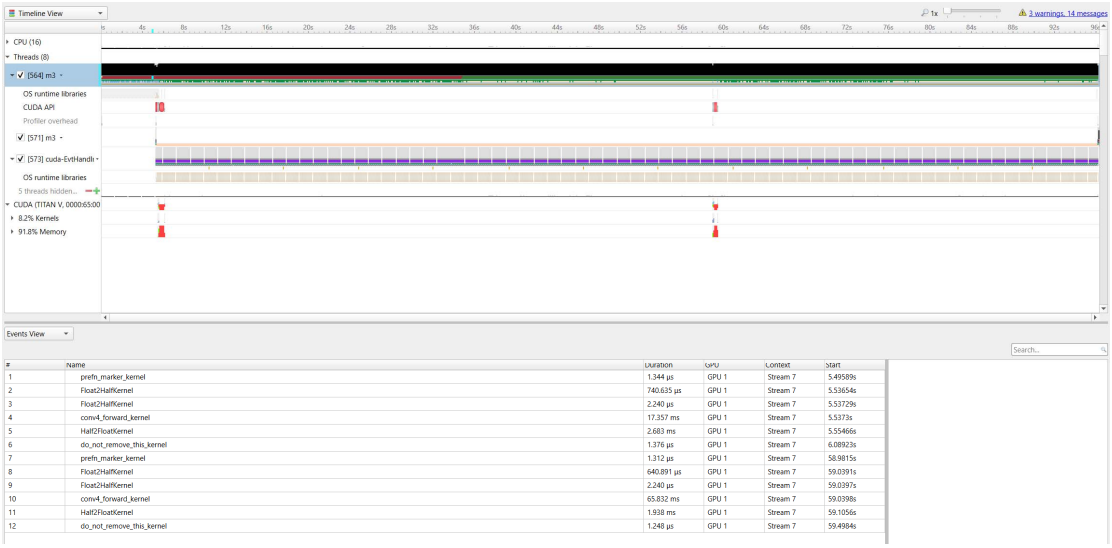


  - second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   74.2      1001976586           8     125247073.3           11928       530680214  cudaMemcpy
   17.9       241196036          14      17228288.3            5319       234359015  cudaMalloc
    6.6        89236258          14       6374018.4             882        65835890  cudaDeviceSynchronize
    1.2        16254992          12       1354582.7            4640        16089731  cudaLaunchKernel
    0.2         2193463           8        274182.9            5189          766120  cudaFree
    0.0           19099           2          9549.5            7438           11661  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   93.3        83188818           2      41594409.0        17356703        65832115  conv4_forward_kernel
    5.2         4620798           2       2310399.0         1938098         2682700  Half2FloatKernel
    1.6         1386006           4        346501.5            2240          740635  Float2HalfKernel
    0.0            2656           2          1328.0            1312            1344  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
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.5       918987384           2     459493692.0       389110560       529876824  [CUDA memcpy DtoH]
    7.5        74946194           8       9368274.3            1536        38601281  [CUDA memcpy HtoD]
```



In addition, we can also improve the test accuracy for this dataset using fixed point (FP16) arithmetic by 0.0002 compared to other methods.

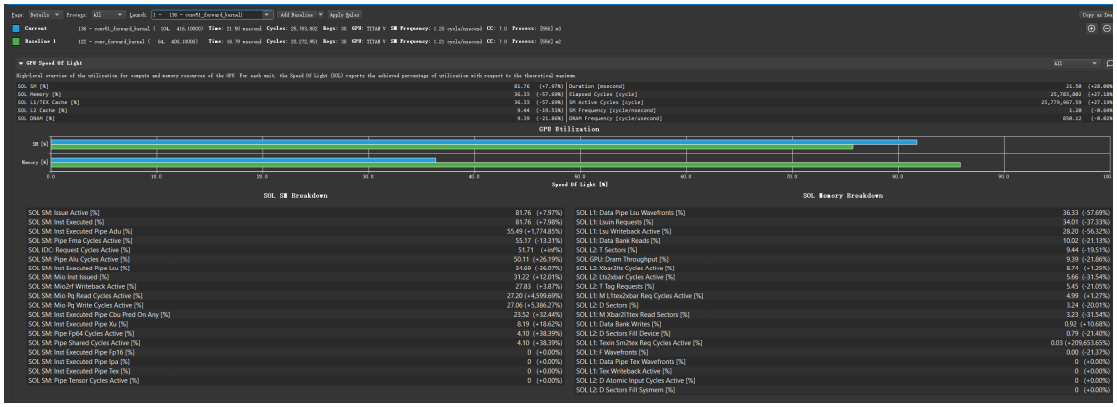
Optimization 5: 1D Shared Memory convolution && Optimization 1

The shared memory in Optimization 2 is a 2D shared memory, here we try if we can use 1D shared memory to improve the shared memory access performance. When compared with the 2D shared memory, the time for the first layer increased but the time for the second layer decreased.

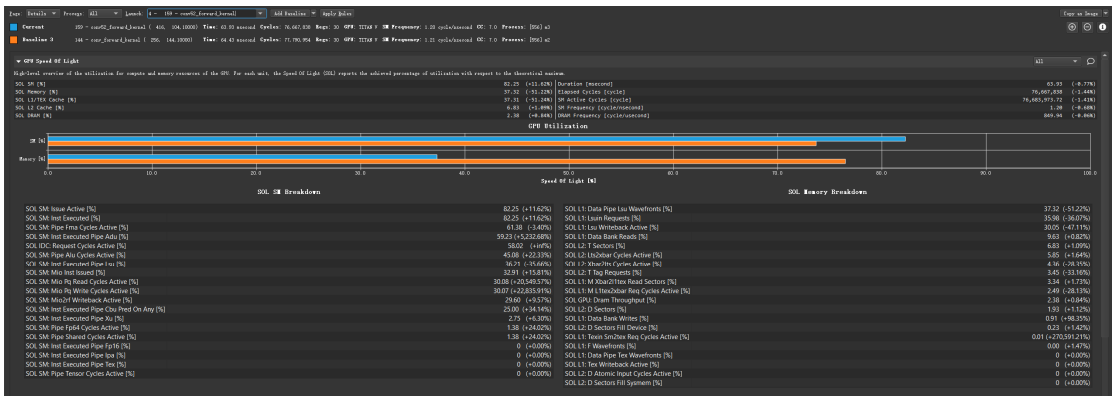
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 630.68 ms
Op Time: 26.9354 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 501.711 ms
Op Time: 75.496 ms

Test Accuracy: 0.8714
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

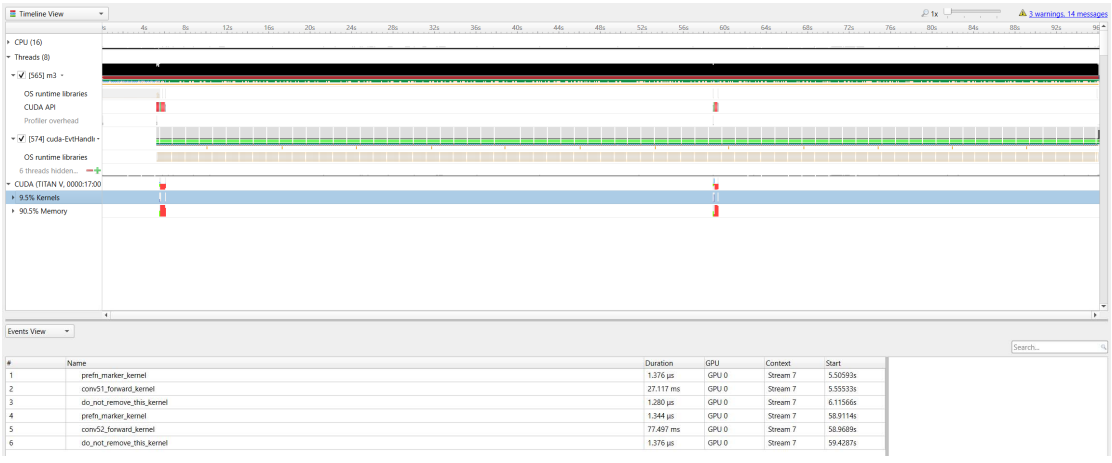


  - second layer



- Output of - nsys profile --stats=true ./m3

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   72.8       992229259           8     124028657.4           15745       531822647  cudaMemcpy
   19.2       262125101          11      23829554.6            6034       251123354  cudaMalloc
    6.6        89746429          11       8158766.3             998        66182500  cudaDeviceSynchronize
    1.1        15242347           9       1693594.1            6456        15102144  cudaLaunchKernel
    0.2         3303061           8        412882.6            4633         1072096  cudaFree
    0.0           18788           2          9394.0            7695           11093  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   73.8        66179274           1     66179274.0        66179274        66179274  conv4_forward_kernel
   23.3        20855325           1     20855325.0        20855325        20855325  conv31_forward_kernel
    2.2         1933754           1      1933754.0         1933754         1933754  Half2FloatKernel
    0.7          641694           2       320847.0            2208          639486  Float2HalfKernel
    0.0            2656           2         1328.0            1280            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
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   92.6       913426065           2     456713032.5       382399434       531026631  [CUDA memcpy DtoH]
    7.4        72567159           8       9070894.9            1472        38404294  [CUDA memcpy HtoD]
```



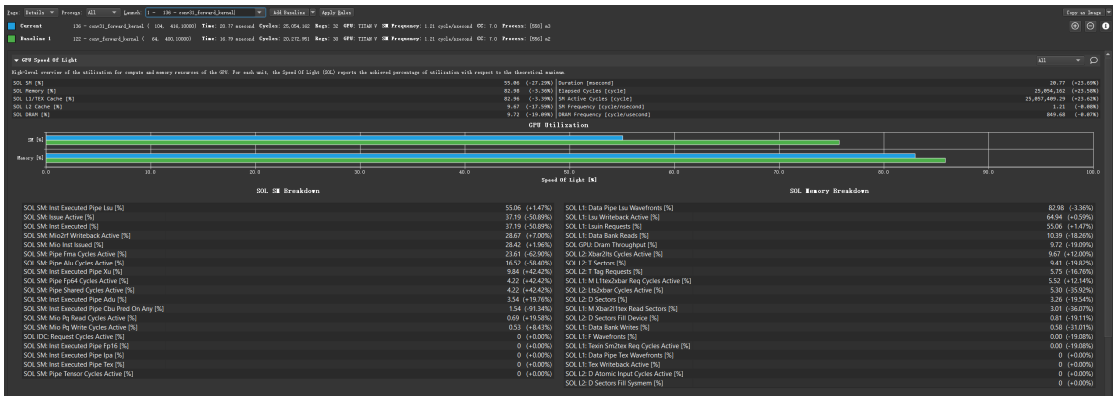
Optimization 6: Multiple kernel implementations for different layer sizes

As we can see from previous optimizations, due to the different sizes of the width, height and number of feature maps of the input/output data, although we are using the same operations for the convolutional layers, the impact on the kernel performance of some specific optimizations may vary among different layers. Thus, this phenomenon gives us the motivation to use different optimization schemes for different layers. After we implemented and analyzed the performance of previous optimizations, we find that the best optimization scheme for the first layer is Optimization 3 (restrict keyword and loop unroll) with a tile width of 26 and the one for the second layer is Optimization 4 (FP 16 arithmetic) with a tile width of 16. As we can see from the figures below, this hybrid scheme is indeed faster than others and require less memory due to the use of half-precision arithmetic.

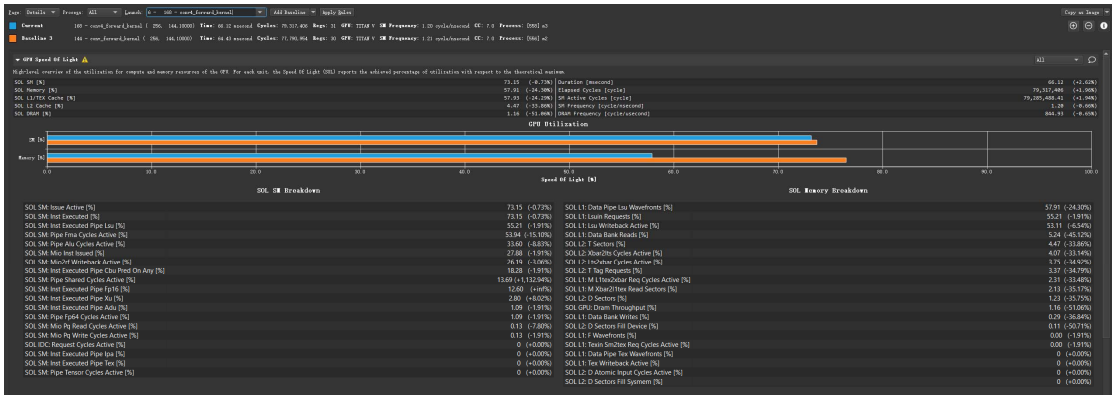
- Output of - /bin/bash -c "./m3"

```
* Running bash -c "./m3"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,4,4,16
Layer Time: 572.825 ms
Op Time: 20.9258 ms
Conv-GPU==
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Layer Time: 473.312 ms
Op Time: 70.1034 ms

Test Accuracy: 0.8716
```

- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
  - first layer

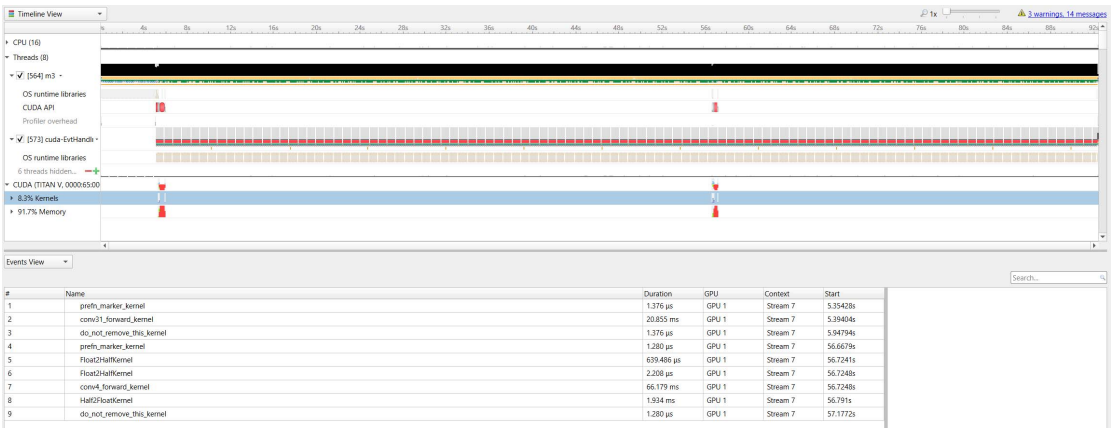


- second layer



- Output of - nsys profile --stats=true ./m3

```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   68.7        66238433           1      66238433.0        66238433        66238433  conv4_forward_kernel
   28.6        27522500           1      27522500.0        27522500        27522500  conv21_forward_kernel
    2.0         1940206           1       1940206.0         1940206         1940206  Half2FloatKernel
    0.7          643290           2        321645.0            2304          640986  Float2HalfKernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
    0.0            2528           2          1264.0            1248            1280  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   91.2        926972500           2     463486250.0       388346139       538626361  [CUDA memcpy DtoH]
    8.8         89674218           8      11209277.2            1568        47983336  [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]
         538932.0               8           67366.0               0.004           288906.0  [CUDA memcpy HtoD]
```



Optimization 7: Sweeping various parameters to find best values

- Output of - /bin/bash -c "./m3"
- Output of - nv-nsight-cu-cli --section '.*' -o analysis_file ./m3
- Output of - nsys profile --stats=true ./m3

Because different optimization scheme may have different best values, we only consider optimizing the shared memory convolution by sweeping the tile size. Due to the limitation of number of threads per block, the maximum of the tile width is 26 (26=sqrt(1024)-7+1). And after we loop through the values below 26, we find that the tile width corresponding to the shortest layer time is 26, which has already been used in the previous shared memory cases.

Summary

All the time consumptions of the above optimizations are listed in the below table for a more intuitive understanding.

Layer	Time (ms)	Baseline	Opt1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6
1	Layer	726.6	661.0	613.4	613.4	632.6	630.7	572.8
	Op	26.0	23.7	27.2	21.0	21.5	27.0	20.9
2	Layer	502.0	490.8	508.1	500.0	498.0	501.7	473.3
	Op	74.5	64.2	78.4	55.8	68.3	75.5	70.1

Milestone 2 – Baseline GPU

1. Show output of rai running your GPU implementation of convolution (including the OpTimes)

To realize this basic kernel, I spent a tremendous amount of time debugging one issue which turned out to be very silly. The problem that leads to the low accuracy (0.1) turned is that I used to represent ‘blockIdx.z’ as ‘bz’ and ‘blockDim.z’ as ‘dz’ but mixed up ‘bz’ and ‘dz’. So the batch dimension wasn't looped over (in parallel) in this problematic case. At the end of the day, the result for running the whole dataset is correct and listed as follow by

- /bin/bash -c "cuda-memcheck ./m2"

As we can see, there is no accuracy error nor memory error.

```
[100%] Built target m2
[100%] Built target final
[100%] Built target m1
[100%] Built target m3
* Running bash -c "cuda-memcheck ./m2"   \\ Output will appear after run is complete.
^[[A^[[A^[[A^[[A^[[A^[[B^[[B========= CUDA-MEMCHECK
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
host_x 0.000000
host_k -0.003229
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
Op Time: 18007.6 ms
Conv-GPU==
host_x 0.000000
host_k 0.005163
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
Op Time: 69526.9 ms

Test Accuracy: 0.8714

========= ERROR SUMMARY: 0 errors
```

2. Demonstrate nsys profiling the GPU execution.

The nsys profiling output is as follows.

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

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
76.5	909015556	8	113626944.5	15652	484421117	cudaMemcpy

14.7	175066418	8	21883302.2	98352	170902717	cudaMalloc
6.9	82114616	8	10264327.0	910	65077737	cudaDeviceSynchronize
1.5	17295511	6	2882585.2	16006	17181641	cudaLaunchKernel
0.4	4844347	8	605543.4	70088	2673695	cudaFree

Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...

CUDA Kernel Statistics (nanoseconds)

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	81972420	2	40986210.0	16896767	65075653	conv_forward_kernel
0.0	2688	2	1344.0	1312	1376	do_not_remove_this_kernel
0.0	2624	2	1312.0	1280	1344	prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
92.1	832698776	2	416349388.0	349011230	483687546	[CUDA memcpy DtoH]
7.9	71568620	6	11928103.3	1536	35903606	[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]

Generating Operating System Runtime API Statistics...

Operating System Runtime API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
33.3	86814287580	882	98428897.5	20619	100208246	sem_timedwait
33.3	86775322869	881	98496393.7	31795	100247224	poll
21.2	55263147610	2	27631573805.0	19731677854	35531469756	pthread_cond_wait
12.1	31509146092	63	500145176.1	500101881	500165522	pthread_cond_timedwait
0.0	101410890	906	111932.5	1011	16240556	ioctl
0.0	18347080	26	705656.9	1153	18275267	fopen
0.0	16196051	9071	1785.5	1018	17608	read
0.0	2727900	97	28122.7	1209	1182760	mmap
0.0	1018883	101	10088.0	4185	23909	open64
0.0	313603	5	62720.6	34775	107733	pthread_create
0.0	196883	1	196883.0	196883	196883	pthread_mutex_lock
0.0	143404	3	47801.3	44263	53539	fgets
0.0	82397	15	5493.1	1281	17030	munmap
0.0	77154	3	25718.0	2498	53069	fopen64
0.0	62682	15	4178.8	2203	8140	write
0.0	40067	7	5723.9	2977	7363	fflush
0.0	29961	5	5992.2	2554	9928	open
0.0	19888	6	3314.7	1019	7189	fclose
0.0	13215	2	6607.5	5803	7412	socket
0.0	10375	2	5187.5	4375	6000	pthread_cond_signal
0.0	7506	1	7506.0	7506	7506	pipe2
0.0	6928	1	6928.0	6928	6928	connect
0.0	4005	2	2002.5	1006	2999	fwrite
0.0	1759	1	1759.0	1759	1759	bind

Generating NVTX Push-Pop Range Statistics...
NVTX Push-Pop Range Statistics (nanoseconds)

3. Include a list of all kernels that collectively consume more than 90% of the program time.

conv_forward_kernel

4. Include a list of all CUDA API calls that collectively consume more than 90% of the program time.

cudaMemcpy, cudaMalloc and cudaDeviceSynchronize. the first two are sufficient and necessary to consume more than 90% of the program time.

5. Include an explanation of the difference between kernels and API call.

- From the experience of the machine problems in this course, the CUDA kernels are usually written by the user while the CUDA API is usually predefined by CUDA libraries.
- Based on the description in 'https://docs.nvidia.com/cuda/cuda-c-programming-guide', kernels are C++ functions executed in parallel by CUDA thread. While CUDA API provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.
- Both the kernel and the API call are launched in the host code. The CUDA APIs called on CPU (a.k.a. host) that transfer data from the host memory are launched synchronously while CUDA kernel is launched asynchronously on the host but will be executed in parallel on the device threads. The syntax '<<<>>>' to launch the kernel belongs to CUDA runtime API.
- From the perspective of profiling, the 'cudaLaunchKernel' in CUDA API Statistics does not include the time of kernel execution while the kernel execution time in 'CUDA Kernel Statistics' is the time that the device takes to run the kernel. The actual kernel execution time in the view of CPU is accounted to 'cudaDeviceSynchronize'.

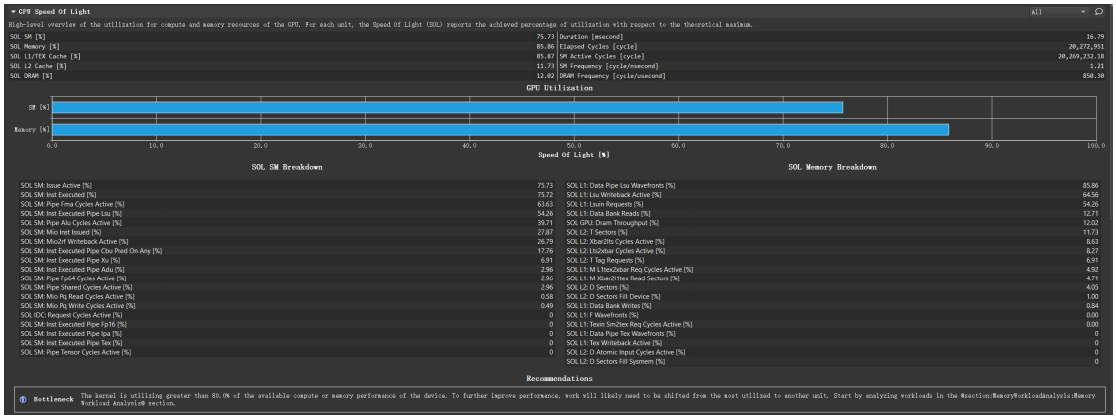
6. Screenshot of the GPU Speed Of Light (SOL) utilization in Nsight-Compute GUI for your kernel profiling data

RAI CLI output

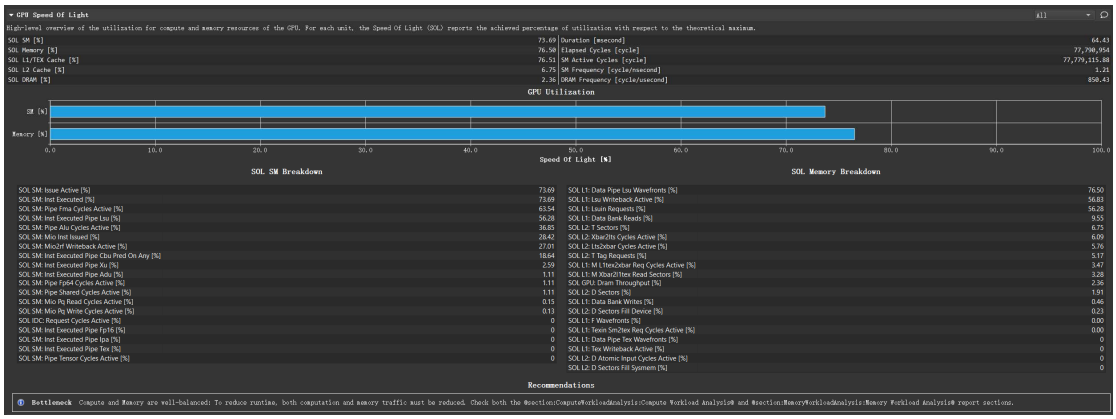
```
* Running ight-cu-cli --section '.*' -o analysis_file ./m2   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
==PROF== Connected to process 556 (/build/m2)
==PROF== Profiling "prefn_marker_kernel()" - 1: 0%....50%....100% - 73 passes
host_x 0.000000
host_k -0.003229
B,M,C,H,W,K is : 10000,4,1,86,86,7
h_out,w_out,w_grid,h_grid,Y is : 80,80,5,5,25
==PROF== Profiling "conv_forward_kernel" - 2: 0%....50%....100% - 74 passes
Op Time: 11662.3 ms
==PROF== Profiling "do_not_remove_this_kernel()" - 3: 0%....50%....100% - 73 passes
Conv-GPU==
==PROF== Profiling "prefn_marker_kernel()" - 4: 0%....50%....100% - 73 passes
host_x 0.000000
host_k 0.005163
B,M,C,H,W,K is : 10000,16,4,40,40,7
h_out,w_out,w_grid,h_grid,Y is : 34,34,3,3,9
==PROF== Profiling "conv_forward_kernel" - 5: 0%....50%....100% - 74 passes
Op Time: 33772.1 ms
==PROF== Profiling "do_not_remove_this_kernel()" - 6: 0%....50%....100% - 73 passes

Test Accuracy: 0.8714

==PROF== Disconnected from process 556
==PROF== Report: /build/analysis_file.ncu-rep
```

Visualize the results in .ncu-rep in Nsight-Compute GUI

1 - 122 - conv_forward_kernel



4 - 144 - conv_forward_kernel



Use 144 - conv_forward_kernel as baseline.