Name: *Wentao Yao*
NetID: *wentaoy4*
Section: *ZJ1/ZJ2*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1, Layer Time1	Op Time 2, Layer Time2	Total Execution Time	Accuracy
100	*0.177546ms, 8.84715ms*	*0.6341ms, 7.94491ms*	*1.238s*	*0.86*
1000	*1.66480ms, 65.8804ms*	*6.1907ms, 52.8546ms*	*10.180s*	*0.886*
10000	*15.793ms, 654.324ms*	*61.149ms, 492.963ms*	*1m39.761s*	*0.8714*

1. **Optimization 1:** ***Tiled shared memory convolution***

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

    I chose to use the Tiled share memory convolution. The reason is that accessing global memory will take a large amount of time. So, using the share memory can reduce the global memory access and reduce the runtime.

    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 works by firstly copy the convolution mask and tiled input data inside global memory into a shared memory. Then in the convolution step, the shared memory will be accessed for computation.
I think this optimization will increase the performance of forward convolution. Because it will reduce the time for accessing global memory.
This optimization is built based on the baseline.

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	*0.172503ms*	*0.889885ms*	*1.275s*	*0.86*
1000	*1.58267ms*	*8.66045ms*	*9.705s*	*0.886*
10000	*15.7197ms*	*86.3424ms*	*1m37.099s*	*0.8714*

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

*The implementation is partially successful. To be specific, the Op time1 has improved while op time2 is longer than the baseline. Based on the analysis of the input, the size of input matrix1 (Height * Width) is larger than matrix2. Since the tiling optimization is for the dimension of Height*Width, then for larger matrix in this dimension, the optimization would be more obvious. While for small matrix, since copying to shared memory still take some time, but the reuse of shared memory is very small, so it will decrease the performance.*
Nsight Compute result:

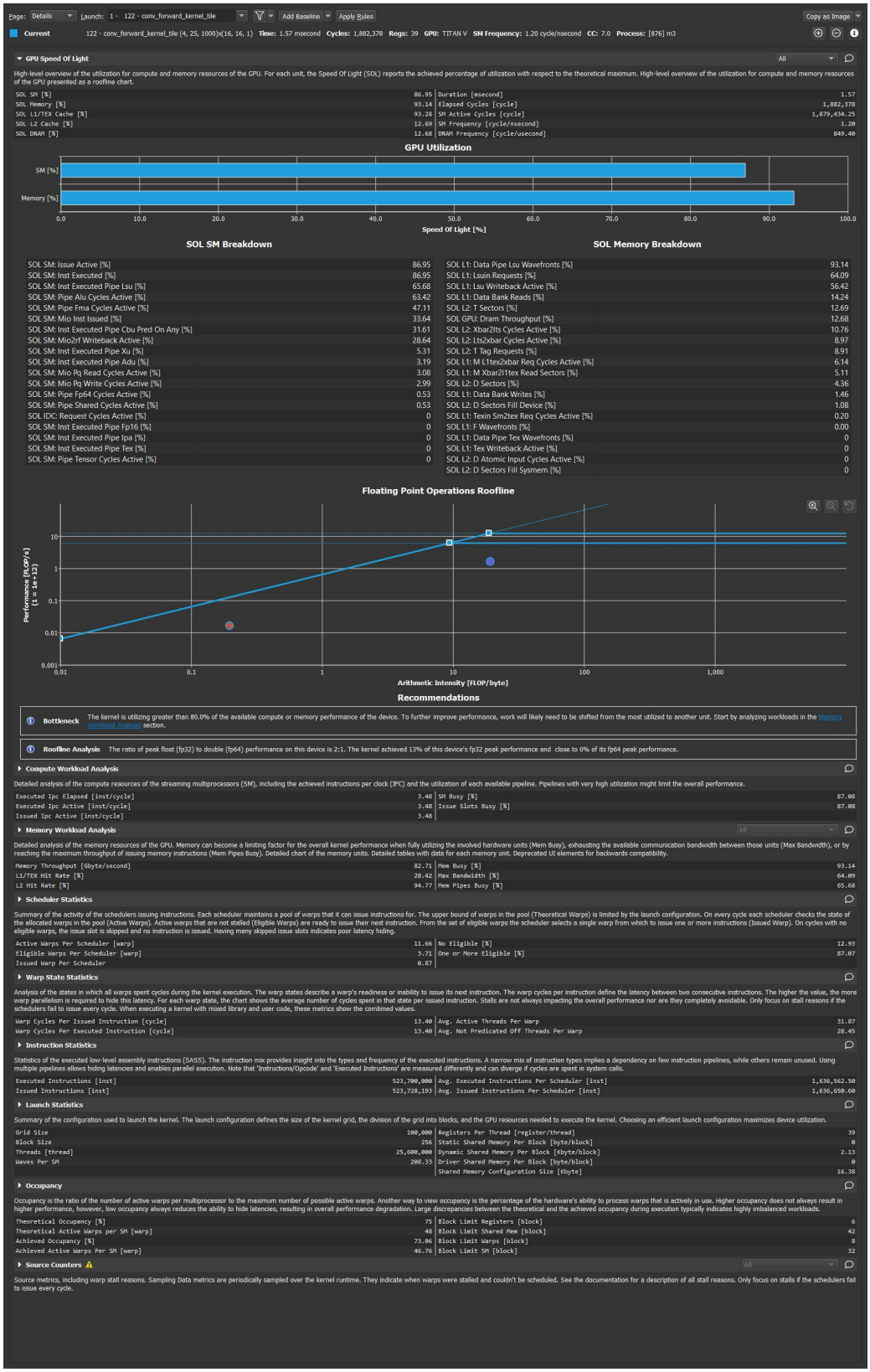

Figure 1Nsight Compute Result: Shared memory

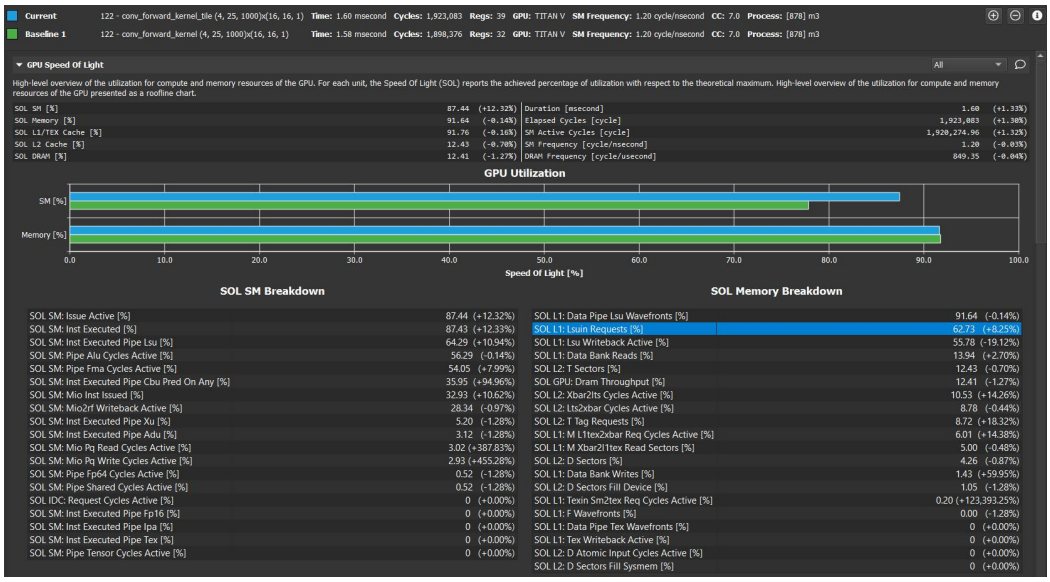

Figure 2Nsight Compute result compared with baseline

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   81.0       652600396           8      81575049.5          212130       576825963  cudaMalloc
   15.6       125698042           8      15712255.3           28218        65673388  cudaMemcpy
    2.3        18149640           6       3024940.0            4527         8661517  cudaDeviceSynchroni
ze
    0.8         6664661           6       1110776.8           27674         6481197  cudaLaunchKernel
    0.4         2822964           8        352870.5           62718         1709633  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   99.9        10233437           2       5116718.5         1570262         8663175  conv_forward_kernel
_tile
    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel

    0.0            2496           2          1248.0            1216            1280  do_not_remove_this_
kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   91.7       113079355           2      56539677.5        48339363        64739992  [CUDA memcpy DtoH]
    8.3        10167966           6       1694661.0            1216         5460925  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  -------------
------------------------------------------------------------
        172250.0               2            86125.0          72250.000           100000.0  [CUDA memcpy
 DtoH]
         53903.0               6             8983.0              0.004            28890.0  [CUDA memcpy
 HtoD]
```

Figure 3Nsys Result: Shared memory

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

1. D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016, ISBN 978-0123814722, Chapter 16.3

2. **Optimization 2: *constant memory for kernel values***

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

I choose to implement constant memory storage for kernel values. The reason is that constant memory can reduce the access to the global device memory, reducing the time for memory access. It's an easy way to improve the 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?

This optimization approach works by copying the mask data from global memory into the constant memory, Using it instead of global memory inside the kernel.
I think it can improve the performance as accessing constant memory is much faster than accessing global memory. It can reduce the time for accessing kernel data.
It implement this optimization based on the baseline.

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	*0.161068ms*	*0.666041ms*	*1.319s*	*0.86*
1000	*1.46983ms*	*6.44835ms*	*9.863s*	*0.886*
10000	*14.3774ms*	*64.6039ms*	*1m37.840s*	*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).

This optimization improve the performance partially. Compare the OP time of this optimization with baseline, OP time1 have improvement while OP time2 is slower than the baseline.
The reason I think for this situation is that probably due to the low cache hit of the constant memory. The constant memory access will reach higher speed when the thread inside the same block is accessing the same address. While in the case of convolution, the rate for this is very low.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   76.3       405969199           6      67661533.2          127035       404573311  cudaMalloc

   21.9       116740916           6      19456819.3           25426        64510978  cudaMemcpy

    1.5         7852320           8        981540.0             969         6367888  cudaDeviceSynchroni
ze
    0.2         1177792           6        196298.7          116934          326741  cudaFree

    0.1          349429           2        174714.5          160382          189047  cudaMemcpyToSymbol

    0.0          182447           6         30407.8           16962           44454  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   99.9         7831438           2       3915719.0         1465457         6365981  conv_forward_kernel

    0.0            2592           2          1296.0            1280            1312  prefn_marker_kernel

    0.0            2560           2          1280.0            1216            1344  do_not_remove_this_
kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   91.6       105198629           2      52599314.5        41594311        63604318  [CUDA memcpy DtoH]

    8.4         9687545           6       1614590.8            1503         5526662  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ------------
------------------------------------------------------------------
        172250.0               2            86125.0          72250.000           100000.0  [CUDA memcpy
 DtoH]
         53903.0               6             8983.0              0.004            28890.0  [CUDA memcpy
 HtoD]
```

Figure 4Nsys Result: Constant Memory

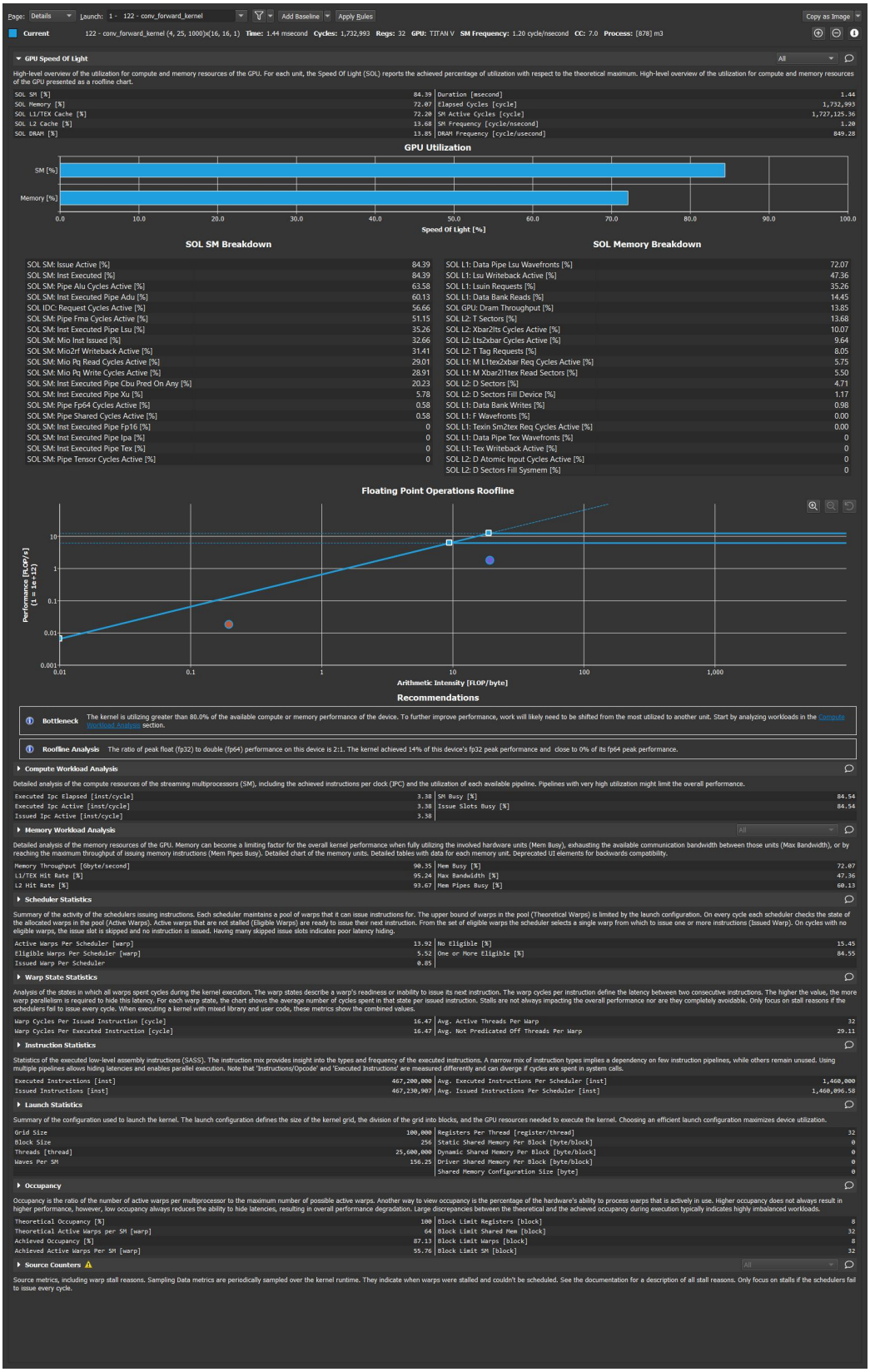

Figure 5Nsight Compute Result: Constant Memory

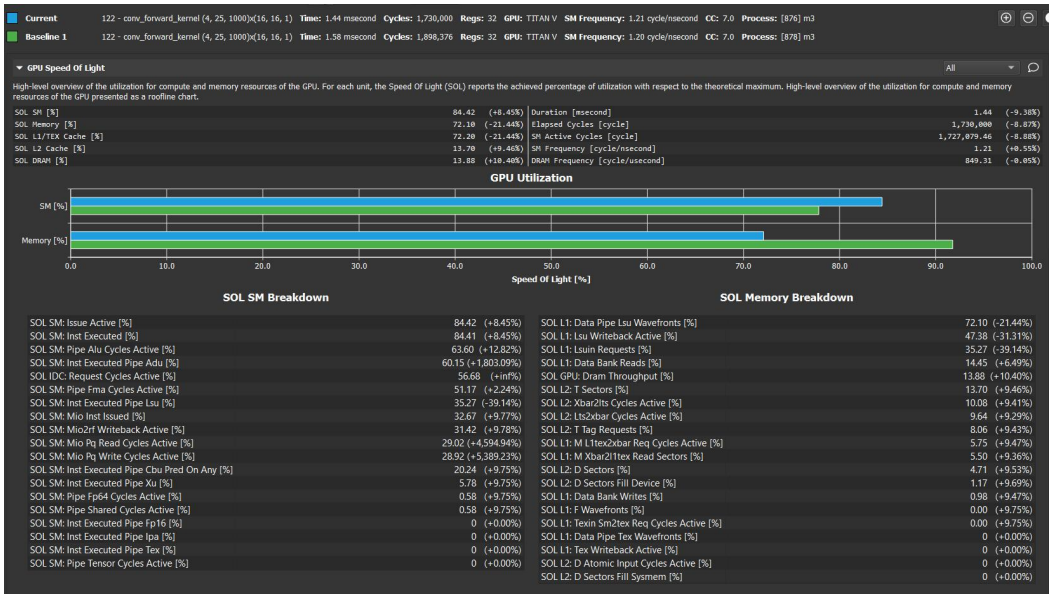

Figure 6Nsight Compute Result compared with baseline

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

   1. *Lecture slide 7: Convolution, Constant Memory and Constant Caching*

3. **Optimization 3: *streams***
   (Delete this section blank if you did not implement this many optimizations.)

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

      Using Streams to overlap the data transfer and computation. The reason to choose it is that overlapping transfer and computation can reduce the total execution time of the model.

   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 work by using N streams to cut all data into N sections, and then overlap the N data section's transfer and computation in a pipeline.
This optimization will not improve the OP time, but it can improve the total execution time of the network. This is due to the workflow of the code, the OP time for this optimization will also count the time for data transfer.
I implement this optimization based on the baseline.

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).
Based on the discussion here: https://campuswire.com/c/G4DE7C13F/feed/1033, due to the special implementation of stream, I move all codes into the prolog function. So the time here is not "real" OP time. Inside the table : (OP time, Layer Time)

Batch Size	Op Time 1, Layer Time	Op Time 2, Layer time2	Total Execution Time	Accuracy
100	*0.0054ms, 10.8084ms*	*0.0049ms, 9.5665ms*	*1.171s*	*0.86*
1000	*0.006ms, 65.1577ms*	*0.005ms, 49.8636ms*	*9.772s*	*0.886*
10000	*0.006ms, 616.154ms*	*0.006ms, 474.974ms*	*1m36.46s*	*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).

This optimization is successful improving the performance. Due to the implementation of stream, we compare the Layer time to the baseline. And results compared with the baseline show that stream can improve the overall layer time of the model.

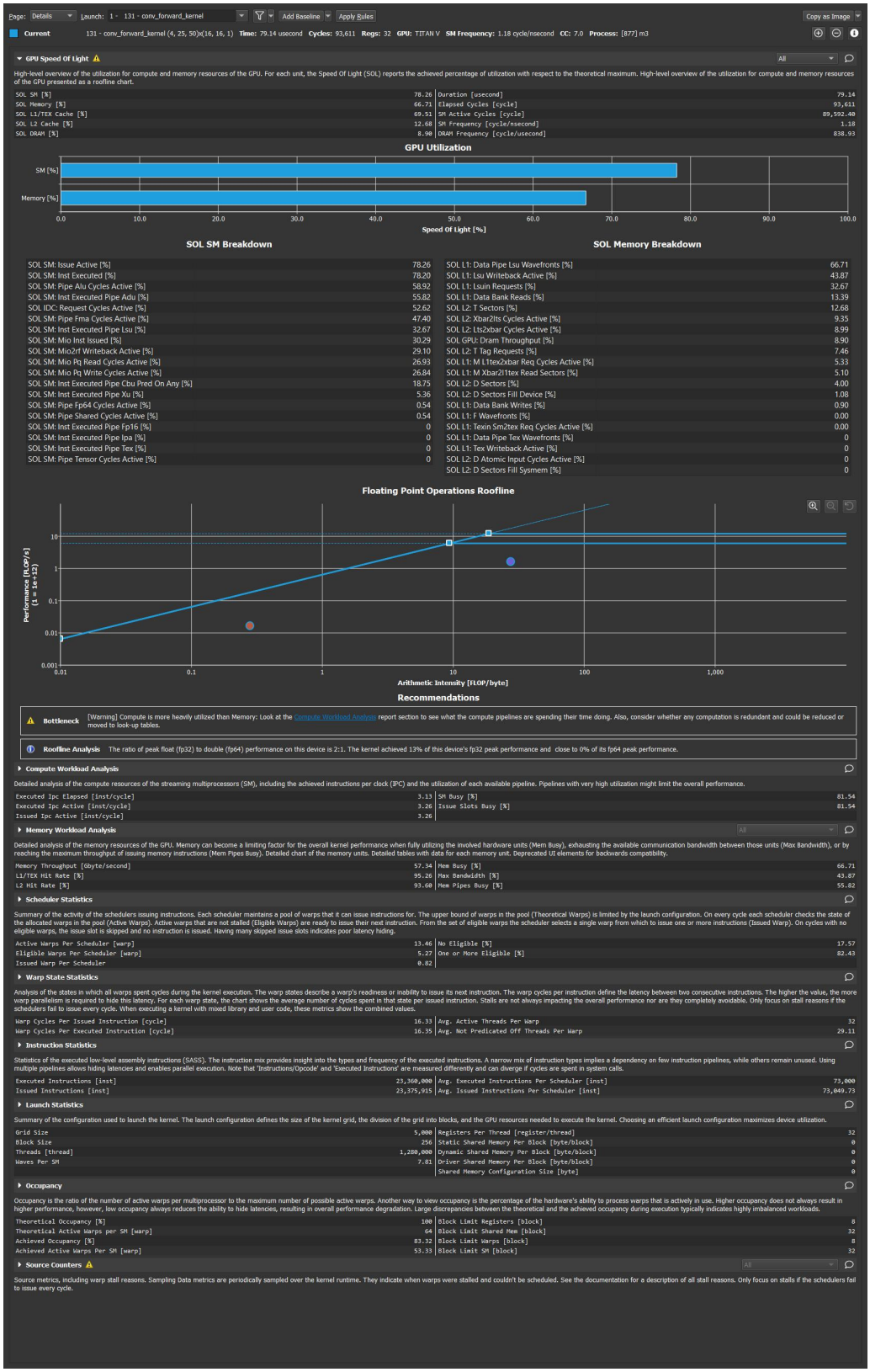

Figure 7Nsight Compute Result: Stream

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  ---------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   58.7       177002602           6      29500433.7          120244       176119109  cudaMalloc

   40.8       123059371          80       1538242.1          185685         3304888  cudaMemcpyAsync

    0.3          962803           6        160467.2           91916          225182  cudaFree

    0.1          437689          44          9947.5            4266           32264  cudaLaunchKernel

    0.0           58636           2         29318.0           12749           45887  cudaMemcpy

    0.0           44419          10          4441.9            1603           19949  cudaStreamCreate

    0.0           33120          10          3312.0            1756            8928  cudaStreamDestroy

    0.0           18775           2          9387.5            7333           11442  cudaMemcpyToSymbol

    0.0           18430           6          3071.7            2152            3920  cudaDeviceSynchroni
ze



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  ---------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   99.9         8830278          40        220757.0           79487          377308  conv_forward_kernel

    0.0            2784           2          1392.0            1280            1504  prefn_marker_kernel

    0.0            2496           2          1248.0            1216            1280  do_not_remove_this_
kernel



CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  ---------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   88.9        72284837          40       1807120.9         1439154         2234889  [CUDA memcpy DtoH]

   11.1         9035558          44        205353.6            1440          247325  [CUDA memcpy HtoD]



CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ------------
--------------------------------------------------------------------
         172250.0              40             4306.0           3612.500             5000.0  [CUDA memcpy
 DtoH]

          53903.0              44             1225.1              0.004             1444.0  [CUDA memcpy
 HtoD]
```

Figure 8Nsys Result: stream

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

   1. *Lecture slide 22: Data Transfer and CUDA Streams*

4. **Optimization 4: Tuning with restrict and loop unrolling**

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

      I choose Turing with restrict and loop unrolling. The reason is that this optimization approach is easy to implement and efficient to improve the performance of model.

   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 work by setting the input memory to kernel with __restrict__ and unroll the loop with #pragma setting by the for loop in convolution step.
This optimization will improve the forward convolution.
Because with the restrict prefix for pointer, the compiler will know that the data read from two pointers will not overlap with each other, it can allow the GPU to use read-only data cache, which can improve the data movement.
With loop unroll, the compiler will unroll the for loop and decrease the time for loading.
I implement this optimization based on baseline code.

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	*0.403641ms*	*1.66792ms*	*1.485s*	*0.86*
1000	*1.57927ms*	*5.69573ms*	*10.397s*	*0.886*
10000	*15.0779ms*	*56.4579ms*	*1m43.693s*	*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).

This optimization successfully improve the performance.
Based on the comparison of Op time between this optimization and baseline, the run time for this optimization is shorter than the baseline.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   67.6       522813839           8      65351729.9           92133       465357843  cudaMalloc
   16.4       126642324           8      15830290.5           64542        69182762  cudaFree
   14.0       107946388           8      13493298.5           19192        57796863  cudaMemcpy
    2.0        15735339           8       1966917.4             987        13638049  cudaDeviceSynchroni
ze
    0.0          173623           6         28937.2           17422           63105  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        11328122           2       5664061.0         1519903         9808219  conv_forward_kernel
    0.0            2880           2          1440.0            1408            1472  prefn_marker_kernel
    0.0            2720           2          1360.0            1280            1440  do_not_remove_this_
kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   91.5        96918861           2      48459430.5        39976843        56942018  [CUDA memcpy DtoH]
    8.5         8977339           6       1496223.2            1184         4799614  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ------------
        172250.0               2            86125.0          72250.000           100000.0  [CUDA memcpy
 DtoH]
         53903.0               6             8983.0              0.004            28890.0  [CUDA memcpy
 HtoD]
```

Figure 9 Nsys Result: loop unrolling

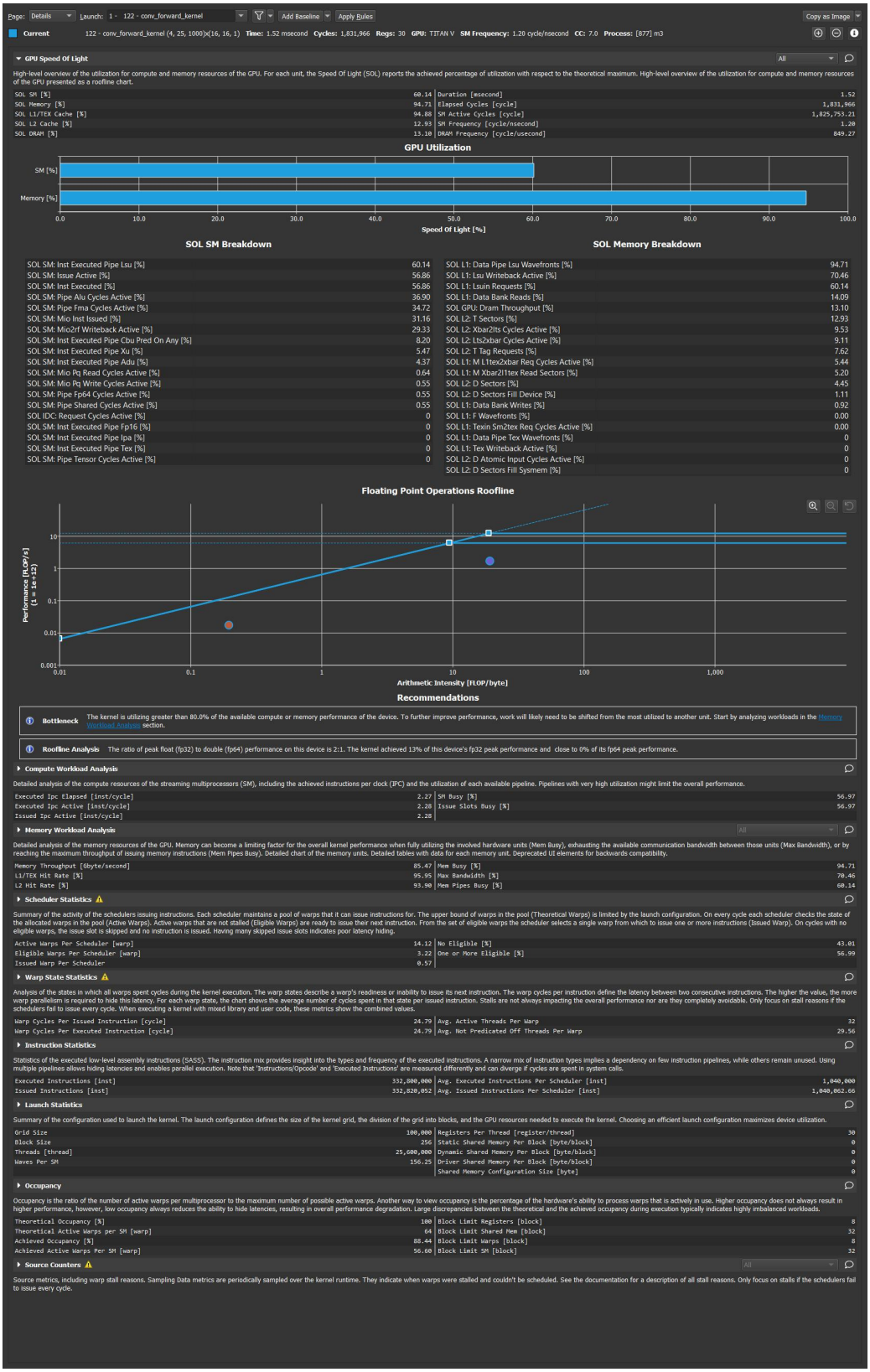

Figure 10 Nsight Compute Result: Loop unroll

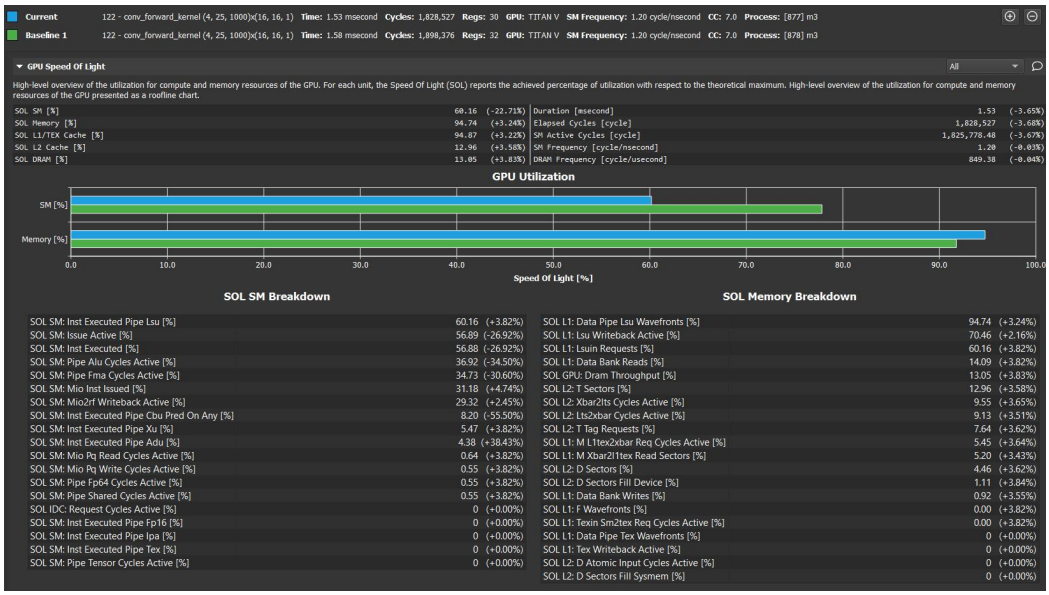

Figure 11 Nsight Compute Result: compare with baseline

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

1. *Strict: https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/#:~:text=While%20you%20can%20sometimes%20explicitly,data%20movement%20to%20your%20kernel.*
2. *Loop Unroll: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#pragma-unroll*

5. **Optimization 5: Fixed point (FP16) arithmetic**

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

I choose to use FP16 optimization.
The reason is that two FP16 number can fit into one 32-bit float number, so FP16 takes smaller memory space for computation, and the throughput is larger.

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 work by firstly converting all float data into half format, send the data into the kernel. Then, inside the kernel, we fit two half data into one single half2 data, then implement the computation. After computation, we store the output into half format, then after the kernel, we transform the half format output to float format.
I think this can improve the performance. Because half2 format can achieve double throughput in computation since two FP16 data can fit to one 32-bit data.
This optimization is synergize with the unroll optimization for the data loading inside the kernel (similar approach).

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	*0.079751ms*	*0.260578ms*	*1.254s*	*0.85*
1000	*0.601692ms*	*2.25819ms*	*9.815s*	*0.854*
10000	*6.51027ms*	*24.8015ms*	*1m41.413s*	*0.8537*

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

The optimization is successful in improving the performance, although the accuracy is influenced by due to half2 (FP16) will lose some part of the original number, which can cause the accuracy loss. But for the performance, the improvement is large. It only has about half of the inference time of previous optimization(constant memory) that it built based on.

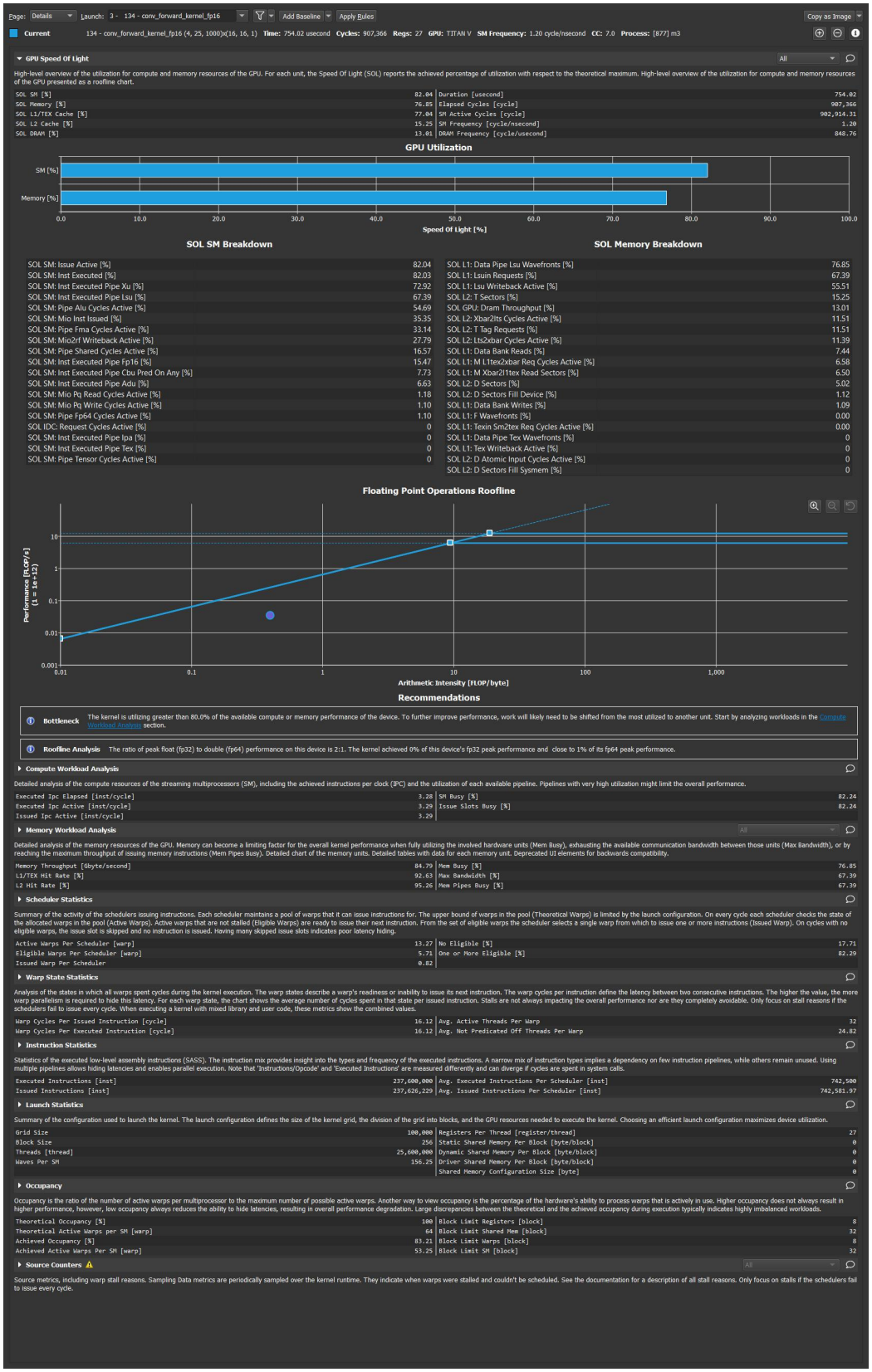

Figure 12 Nsight Compute Result: FP16

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   62.4       186019499          14      13287107.1            5909       184642780  cudaMalloc

   35.9       106908334           8      13363541.8           12971        57676224  cudaMemcpy

    1.4         4197403          12        349783.6            2748         2781432  cudaDeviceSynchroni
ze
    0.3          810820           8        101352.5            6474          173259  cudaFree

    0.1          160618          12         13384.8            4260           26325  cudaLaunchKernel




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   85.2         3551875           2       1775937.5          772858         2779017  conv_forward_kernel
_fp16
   11.2          465341           2        232670.5          195391          269950  fp16_float

    3.6          148319           4         37079.8            1664           77055  float_fp16

    0.1            2784           2          1392.0            1280            1504  prefn_marker_kernel

    0.1            2368           2          1184.0            1184            1184  do_not_remove_this_
kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   91.5        95902886           2      47951443.0        39117487        56785399  [CUDA memcpy DtoH]

    8.5         8960626           6       1493437.7            1184         4800502  [CUDA memcpy HtoD]



CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ------------
------------------------------------------------------------
         172250.0               2            86125.0          72250.000           100000.0  [CUDA memcpy
 DtoH]
          53903.0               6             8983.0              0.004            28890.0  [CUDA memcpy
 HtoD]
```

Figure 13 Nsys Result: FP16

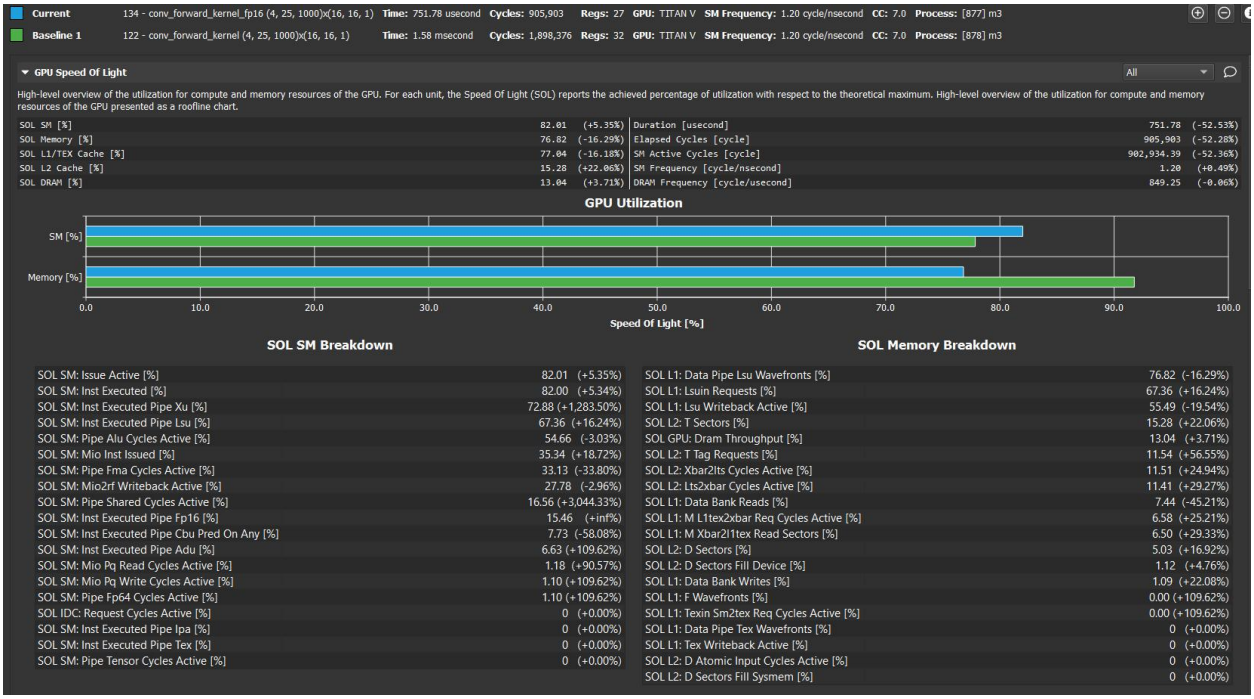

Figure 14 Nsight Compute Result: compare with baseline

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

1. *Half/half2 precision computation: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html#group__CUDA__MATH__INTRINSIC__HALF*

6. **Optimization 6: atomics for input channel reduction**

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

I choose the input channel reduction with atomics operation.
The reason why I choose this is that we can fully utilize the parallel capability in channel dimension through that approach.

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 works by adding the Channel dimension to the block dim, then accumulate each channel's result through atomic add operation.
I think this optimization would increase the performance of forward convolution. Because we can fully utilize the parallel capability of the device by adding the channel dimension.
I implement this optimization based on the baseline.

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	*0.173734ms*	*0.773081ms*	*1.087s*	*0.86*
1000	*1.60302ms*	*7.56941ms*	*9.676s*	*0.886*
10000	*15.6843ms*	*75.801ms*	*1m34.562s*	*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).

This optimization doesn't successfully improve the performance. Compare the result shown here with the baseline, it's clear than OP time is larger than the baseline result. Based on my observation, I find that the input channel is very small, first operation is 1 channel, and second operation has 4 channels. In that case, the parallel computation through the channel dimension isn't obvious. While at the same time, since the atomic operation may meet with situation that threads write in the same address, which may cost additional time.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  ---------------  ---------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   59.0       170627231           8      21328403.9           71288       169653460  cudaMalloc
   37.5       108398320           8      13549790.0           20134        58828985  cudaMemcpy
    3.2         9239597           6       1539932.8            2903         7632635  cudaDeviceSynchroni
ze
    0.4         1040137           8        130017.1           61060          182378  cudaFree
    0.0          134318           6         22386.3           16963           27661  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  ---------------  ---------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   99.9         9223761           2       4611880.5         1592915         7630846  conv_forward_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
    0.0            2464           2          1232.0            1216            1248  do_not_remove_this_
kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  ---------------  ----------  --------------  --------------  --------------  --------------------
------------------------------------------------------------
   91.6        97384664           2      48692332.0        39447467        57937197  [CUDA memcpy DtoH]
    8.4         8960595           6       1493432.5            1216         4799671  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ------------
--------------------------------------------------------------------
         172250.0               2            86125.0          72250.000           100000.0  [CUDA memcpy
 DtoH]
          53903.0               6             8983.0              0.004            28890.0  [CUDA memcpy
 HtoD]
```

Figure 15 Nsys Result: atomic operations

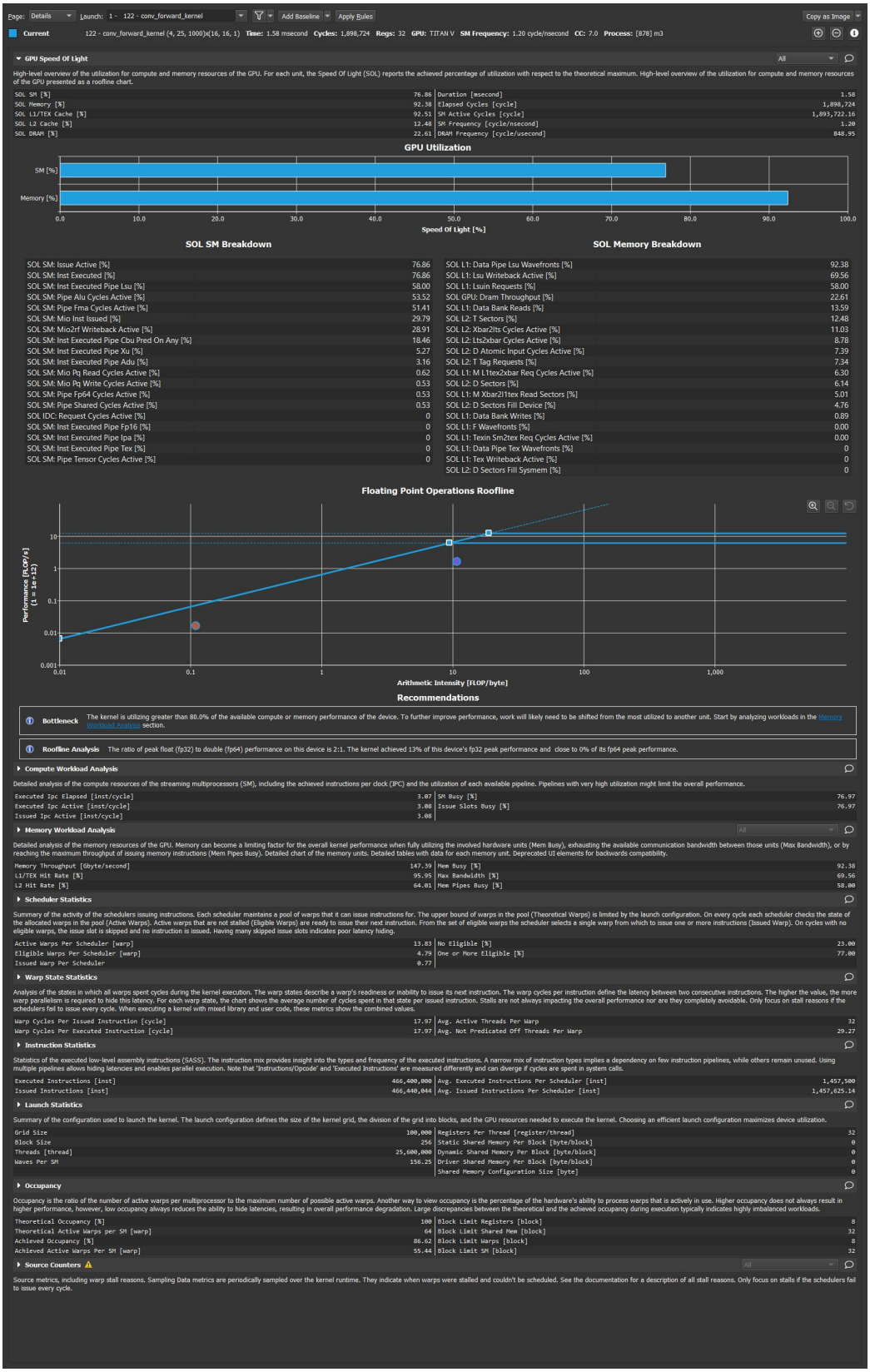

Figure 16 Nsight Compute result: atomic operations

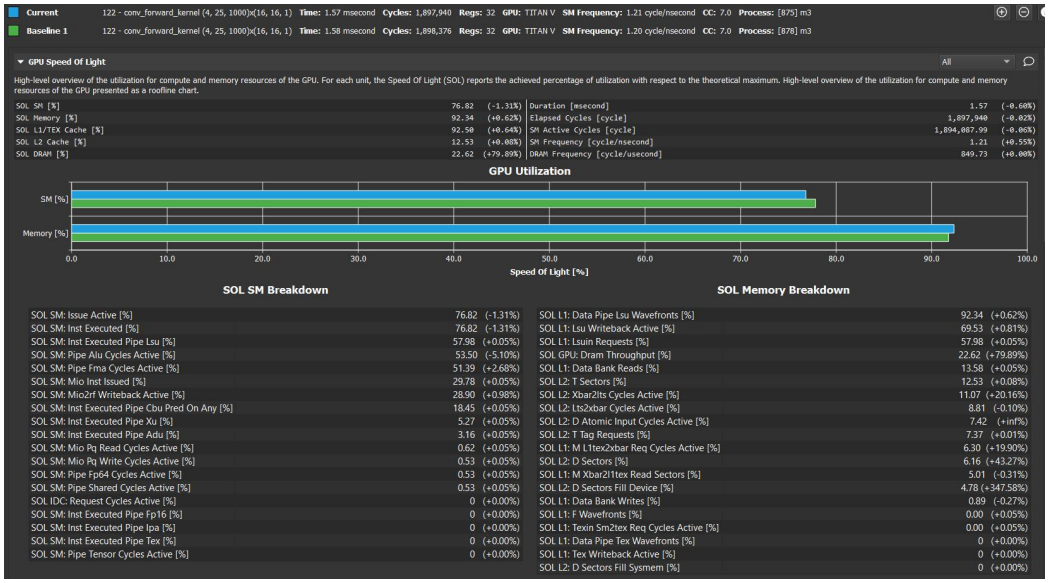

Figure 17 Nsight Compute Result compare with baseline

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

1. *Lecture 18 slide: Atomic Operations and Histogramming*