Name: *Dingkun Wang*
NetID: *dingkun2*
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	Op Time 2	Total Execution Time	Accuracy
100	*0.185017ms*	*0.640771ms*	*0m1.154s*	*0.86*
1000	*1.61016ms*	*6.12609ms*	*0m9.680s*	*0.886*
10000	*15.7674ms*	*61.0281ms*	*1m34.784s*	*0.8714*

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

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

    I choose to implement weight matrix in constant memory because I think it is the easiest one, so I start with this one.

    b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

    This optimization works since we reduce the memory bandwidth by replacing global memory access by constant memory access. I think it would increase performance since we save the time to access global memory. No.

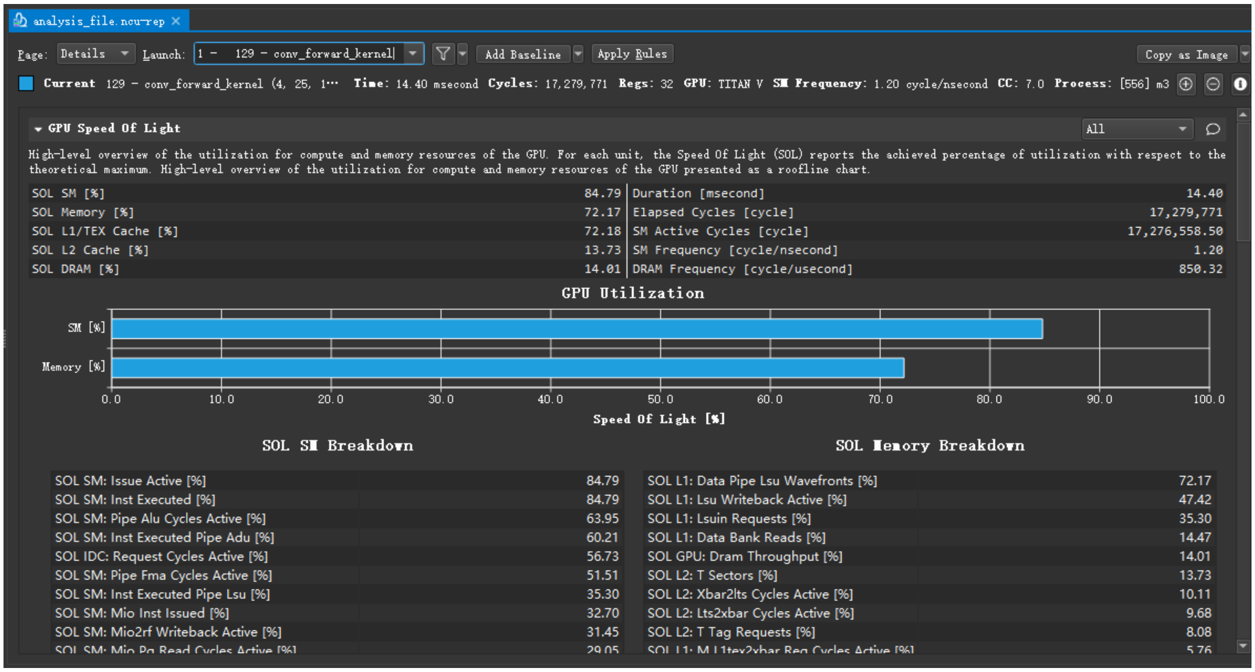
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.159271ms*	*0.662277ms*	*0m1.148s*	*0.86*
1000	*1.45623ms*	*6.44311ms*	*0m9.920s*	*0.886*
10000	*13.0384ms*	*64.6189ms*	*1m40.277s*	*0.8714*

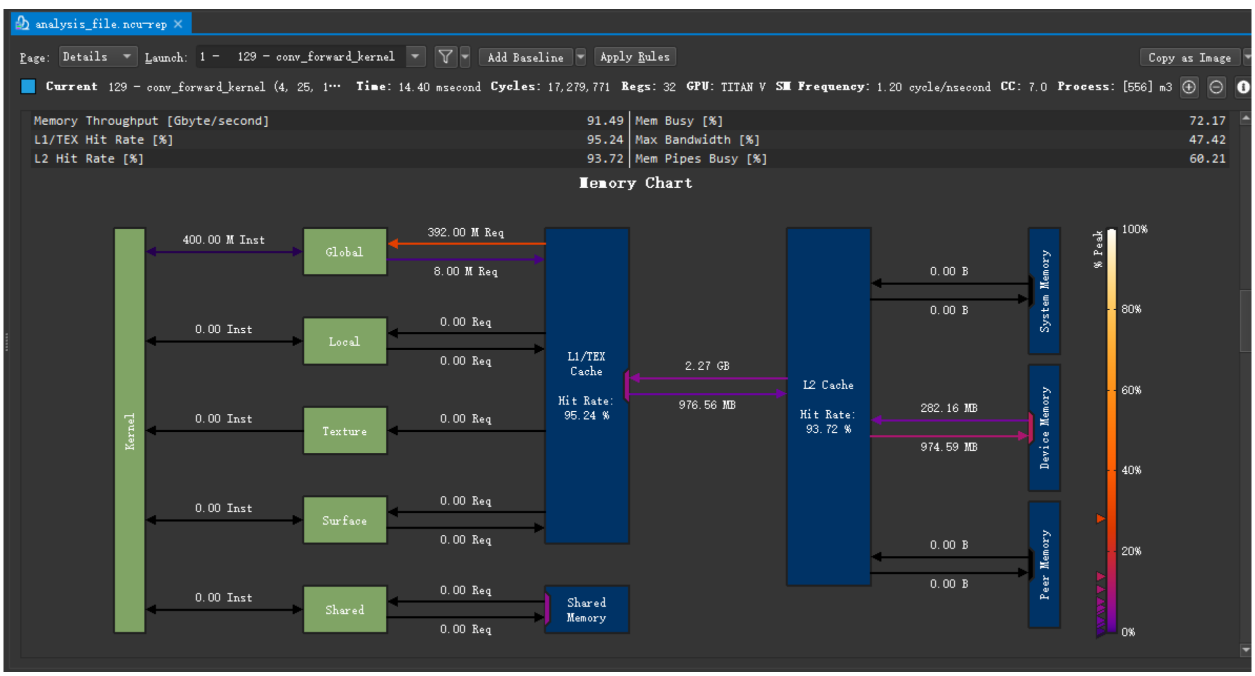
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

It successfully reduces the OP time1 but it slightly increases OP time2, the overall performance does not improve much. Because it only saves time to access global memory slightly due to the small size of masks.
We can see that from profiling the conv_forward_kernel time is 78.23ms, which is nearly unchanged compared to the baseline.
The SM and memory utilization are better than the baseline, which is expected.

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

Time(%)      Total Time       Calls         Average        Minimum        Maximum  Name

   78.6      1099063121           8    137382890.1          12006      596228855  cudaMemcpy
   14.4       201318568           8     25164821.0          70186      197961450  cudaMalloc
    5.6        78255292           6     13042548.7           2531       63603643  cudaDeviceSynchronize
    1.2        16739554           6      2789925.7          15165       16623627  cudaLaunchKernel
    0.2         2549787           8       318723.4          62972         901473  cudaFree
    0.0           21223           2        10611.5           7145          14078  cudaMemcpyToSymbol


Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average        Minimum        Maximum  Name

  100.0        78233305           2     39116652.5       14630865       63602440  conv_forward_kernel
    0.0            2816           2         1408.0           1376           1440  prefn_marker_kernel
    0.0            2784           2         1392.0           1376           1408  do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average        Minimum        Maximum  Name

   91.8      1004214219           2    502107109.5      408854203      595360016  [CUDA memcpy DtoH]
    8.2        89710747           8     11213843.4           1184       48148188  [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]
```



s



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

Course slides.

2. **Optimization 2:** ***Tiled shared memory convolution (2 pts)***

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

    Tiles shared memory convolution. I choose it because it was taught in the lecture and it is not hard to implement that method.

    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?

    We use shared memory to increase memory reuse so that we can save time to access global memory which is rather slow. I think this method would improve the performance due to the reason just mentioned. It synergizes with weight matrix in constant memory optimization.

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

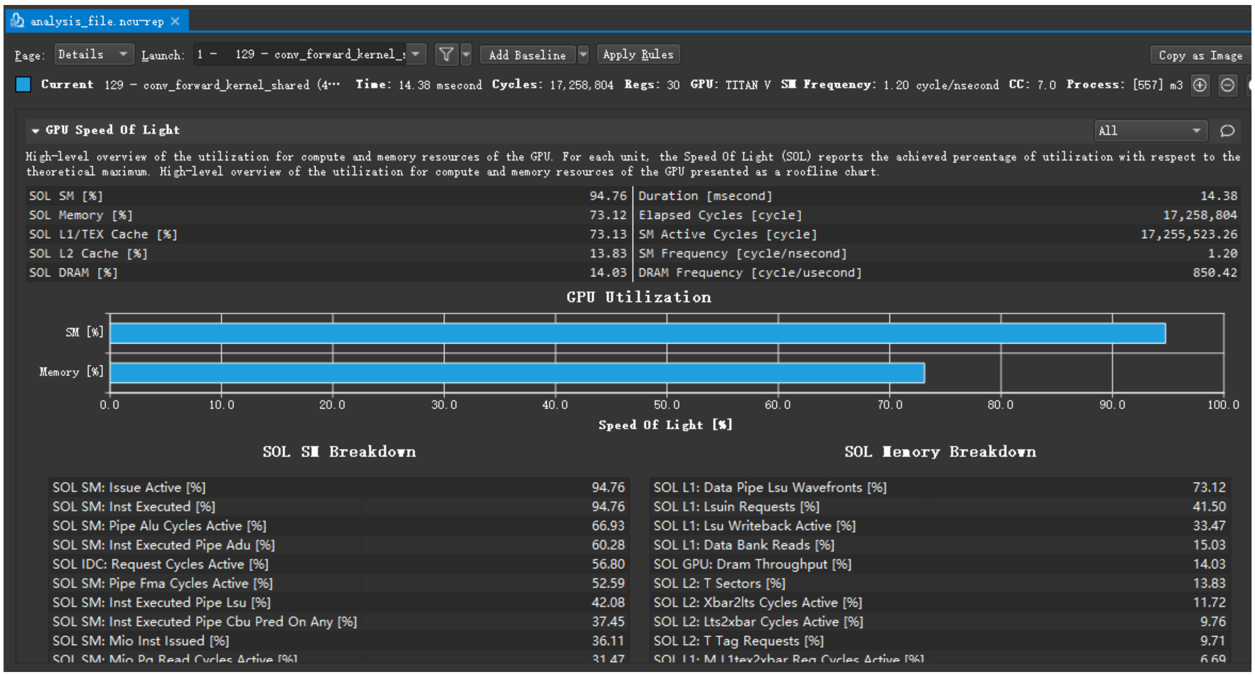
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.160671ms*	*0.632058ms*	*0m1.251s*	*0.86*
1000	*1.45417ms*	*6.09246ms*	*0m10.239s*	*0.886*
10000	*14.455ms*	*61.0096ms*	*1m37.067s*	*0.8714*

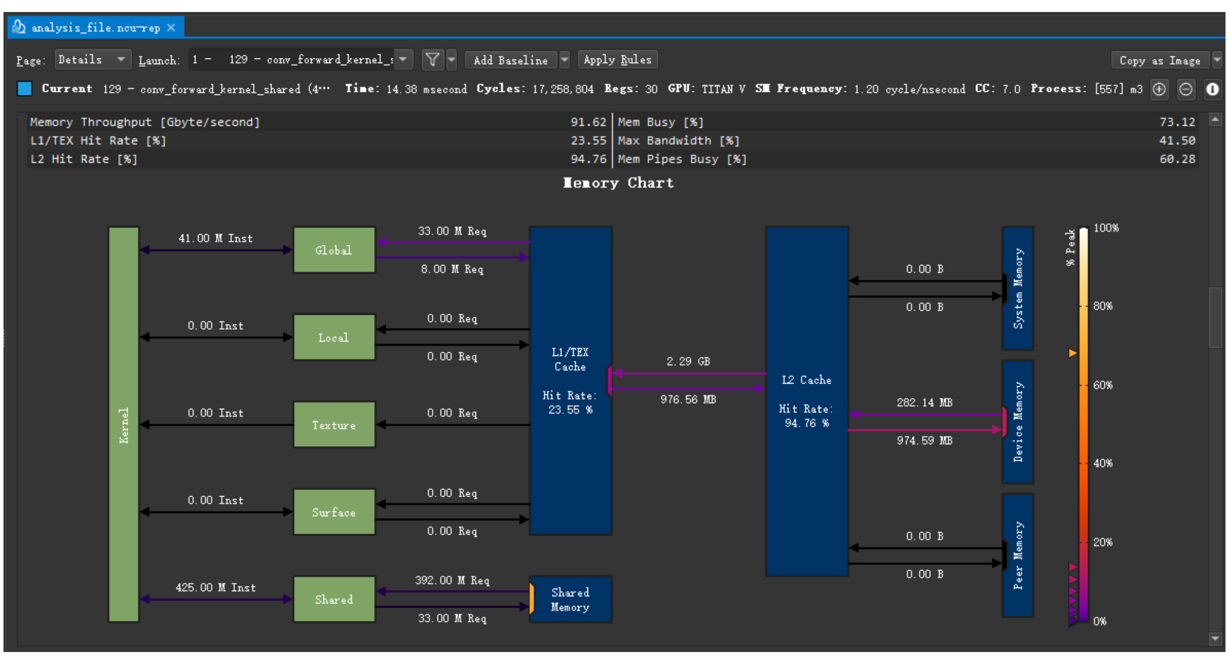
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Not really. We can see that OP time2 has reduced a little bit, but not too much. The overall kernel time is nearly the same (or say reduces slightly) with the implementation without tiling method. I think the that may come from the fact that the tile size is not that appropriate so the improvement is not obvious. The use of shared memory may incur other overheads too.

```
Time(%)      Total Time      Calls         Average         Minimum         Maximum  Name
-------  --------------  ---------  --------------  --------------  --------------  --------------------
--------
   60.8       185366129          8      23170766.1           70673       184332146  cudaMalloc
   36.4       110886817          8      13860852.1           18361        59737418  cudaMemcpy
    2.5         7592336          6       1265389.3            2927         6113097  cudaDeviceSynchronize
    0.3         1012964          8        126620.5           56589          238004  cudaFree
    0.0          121972          6         20328.7           14517           25009  cudaLaunchKernel
    0.0           18305          2          9152.5            7325           10980  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time  Instances         Average         Minimum         Maximum  Name
-------  --------------  ---------  --------------  --------------  --------------  --------------------
--------
   99.9         7574220          2       3787110.0         1463030         6111190  conv_forward_kernel_shared
    0.0            2720          2          1360.0            1344            1376  prefn_marker_kernel
    0.0            2624          2          1312.0            1216            1408  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
--------
   93.1       101160679           2      50580339.5        42394364        58766315  [CUDA memcpy DtoH]
    6.9         7469548           8        933693.5            1216         3987172  [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]
          53916.0               8            6739.0             0.004           28890.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
```

We can also see from memory chart that the access to global memory significantly decreases.





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

Course slides and textbook. Also Campuswire and Stackoverflow.

3. **Optimization 3: *Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (1 pt)***

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

I try to find best parameters to achieve best performance. I change the TILE_WIDTH parameter from 16 to other values from previous optimization and I find that 19 is a goof choice. I choose this because the previous method's performance is not ideal and I guess that is due to poor parameter values.

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 performance is related to the TILE_WIDTH since the width and height of the input array may not divide TILE_WIDTH completely. I try several choices and compare the results. I think it will increase the performance for sure. It synergizes with the tile shared convolution and weight matrix in constant memory.

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

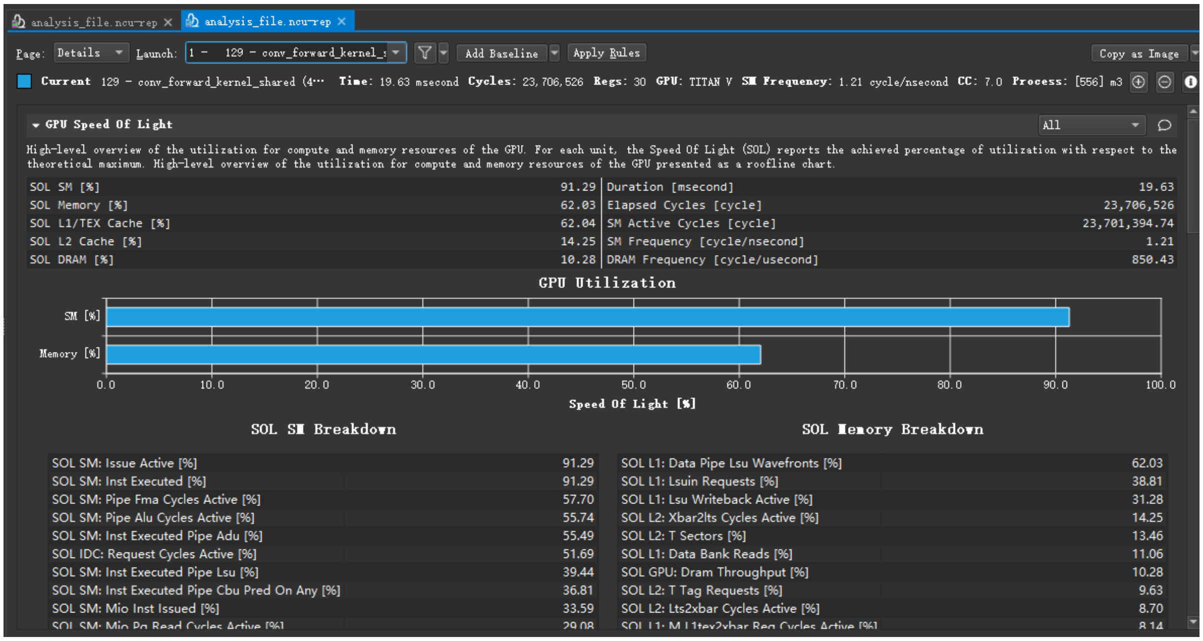
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.212997ms*	*0.489966ms*	*0m1.146s*	*0.86*
1000	*1.99331ms*	*4.72772ms*	*0m9.965s*	*0.886*
10000	*17.754ms*	*42.1521ms*	*1m40.188s*	*0.8714*

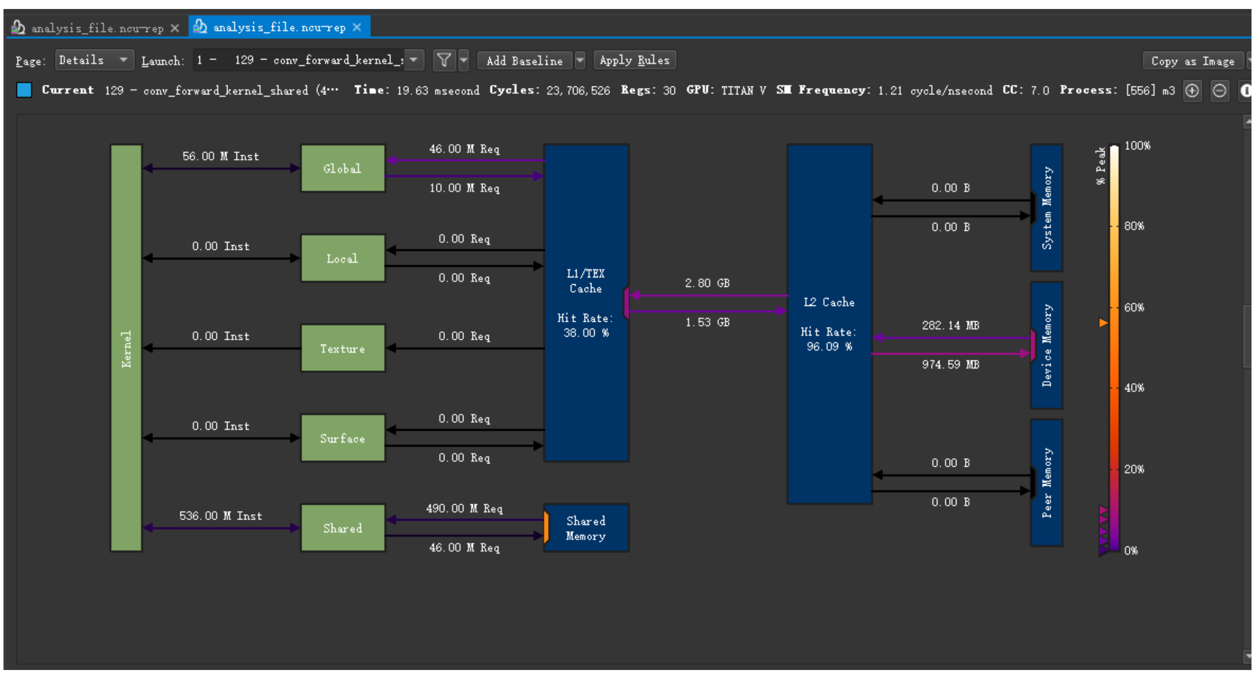
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes. Because I choose the best parameter which makes the performance better.

According to the profiling results and the previous ones without changing the parameters, we can see that the kernel running time reduces a lot, from previous 75.74ms to 66.98ms.

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
   59.6       170903618           8      21362952.2           64732       169922296  cudaMalloc
   37.7       108201909           8      13525238.6           16851        58537672  cudaMemcpy
    2.3         6713409           6       1118901.5            2916         4698529  cudaDeviceSynchronize
    0.3          898373           8        112296.6           55200          231145  cudaFree
    0.0          126563           6         21093.8           12742           27013  cudaLaunchKernel
    0.0           16608           2          8304.0            6813            9795  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
   99.9         6697713           2       3348856.5         2000370         4697343  conv_forward_kernel_shared
    0.0            2656           2          1328.0            1280            1376  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
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
   93.3        98854482           2      49427241.0        41268418        57586064  [CUDA memcpy DtoH]
    6.7         7128623           8        891077.9            1216         3822854  [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]
          53916.0               8             6739.0              0.004            28890.0  [CUDA memcpy HtoD]
```





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

Course slides and videos.

4. **Optimization 4:** ***Input channel reduction: atomics (2 pts)***

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

   Input channel reduction: atomics. I choose this because I just learned atomics optimization in the lecture and I think it is not hard to implement.

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?

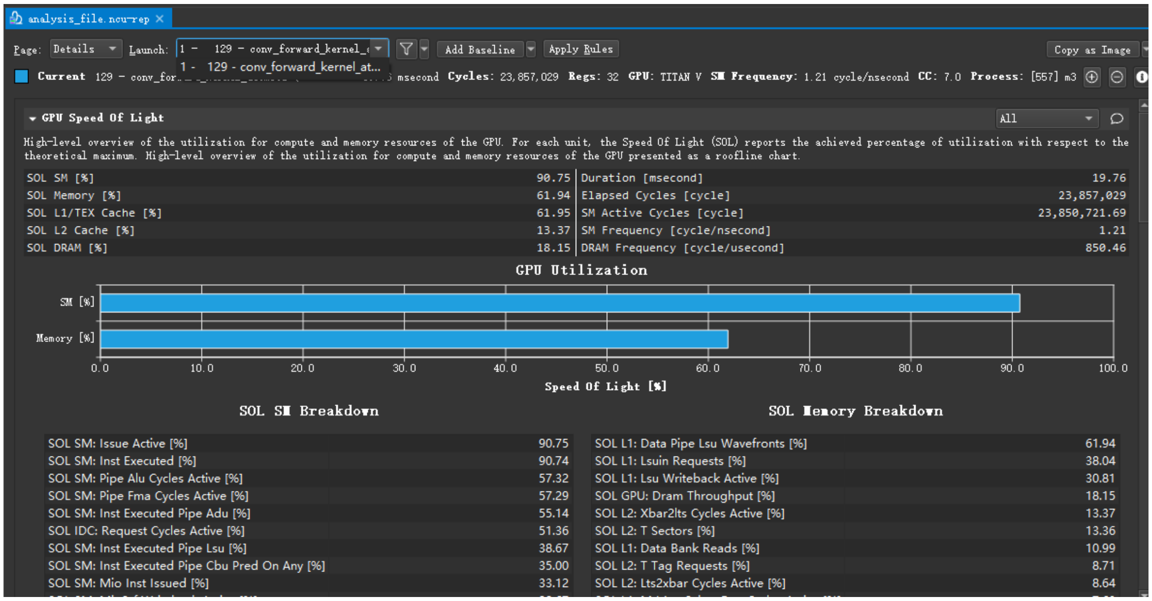
   We can let each thread to deal with one channel and all of the threads contributes to the corresponding feature map output. With atomic add, we can avoid race condition and make the most use of the threads. (Increase number of threads and thus increase parallelism). I think it can increase the performance because all channels can work at the same time. It synergizes with weight matrix in constant memory and tile shared memory convolution.

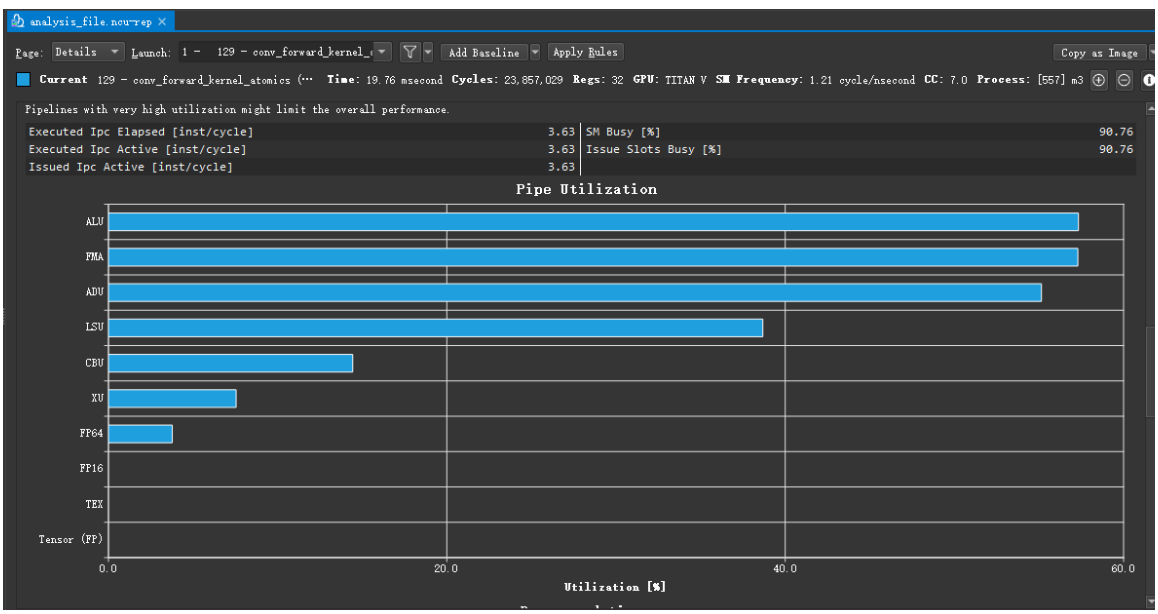
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.16114ms*	*0.777276ms*	*0m1.159s*	*0.86*
1000	*1.46824ms*	*7.67412ms*	*0m9.837s*	*0.886*
10000	*14.4895ms*	*76.7997ms*	*1m35.675s*	*0.8714*

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

No, the total OP time increases as shown in the profiling results. We can see that the SOL decrease for both SM and memory. I think the reason why it does not work is that although it can increase parallelism, it invokes complicated function atomicAdd() which has extra overheads thus takes more time.





Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
60.9	192983291	8	24122911.4	64815	192002823	cudaMalloc
35.8	113376020	8	14172002.5	19066	60780087	cudaMemcpy
2.9	9206459	6	1534409.8	2855	7725549	cudaDeviceSynchronize
0.3	1092897	8	136612.1	58179	297096	cudaFree
0.0	136904	6	22817.3	14678	34315	cudaLaunchKernel
0.0	19044	2	9522.0	7681	11363	cudaMemcpyToSymbol

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	9191073	2	4595536.5	1467158	7723915	conv_forward_kernel_atomics
0.0	2720	2	1360.0	1344	1376	prefn_marker_kernel
0.0	2560	2	1280.0	1248	1312	do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
93.2	103500797	2	51750398.5	43714517	59786280	[CUDA memcpy DtoH]
6.8	7596045	8	949505.6	1216	4068901	[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]
53916.0	8	6739.0	0.004	28890.0	[CUDA memcpy HtoD]

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

Course slides and textbook.

5. **Optimization 5: *Input channel reduction: tree (3 pts)***

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

Input channel reduction: tree. Because the tree reduction is also operated on channels as the previous one and I learn the technique in the lecture so it will not be hard to implement.

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?

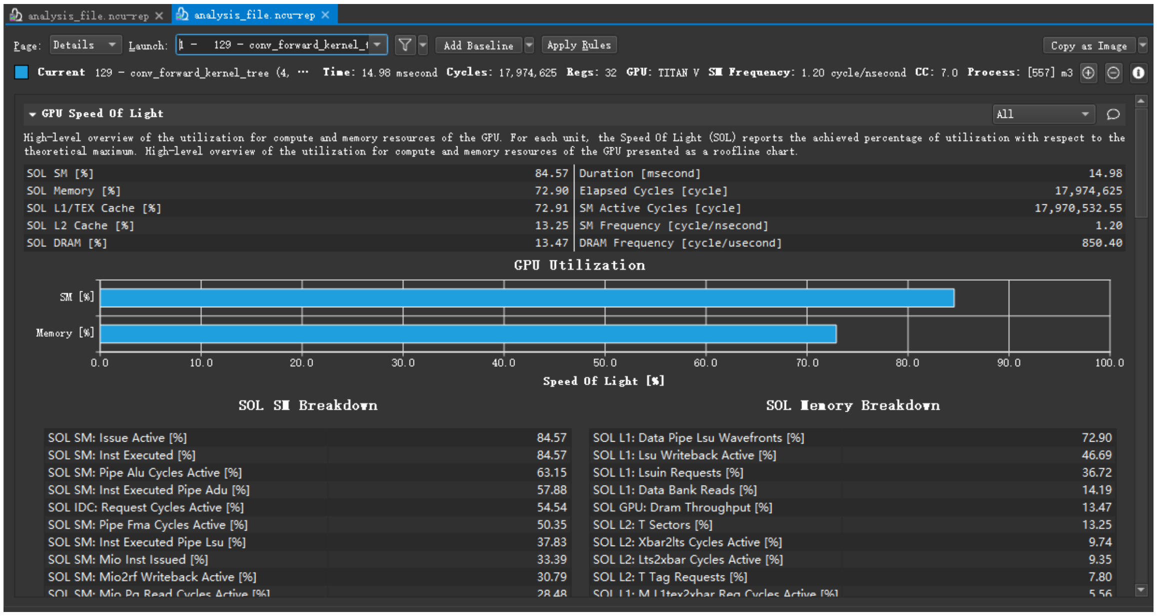
Like atomics, we will tree reduction to make each thread deal with a channel and then use reduction to increase parallelism. I think it will since it can deal with many channels at the same time. It synergizes with weight matrix in constant memory.

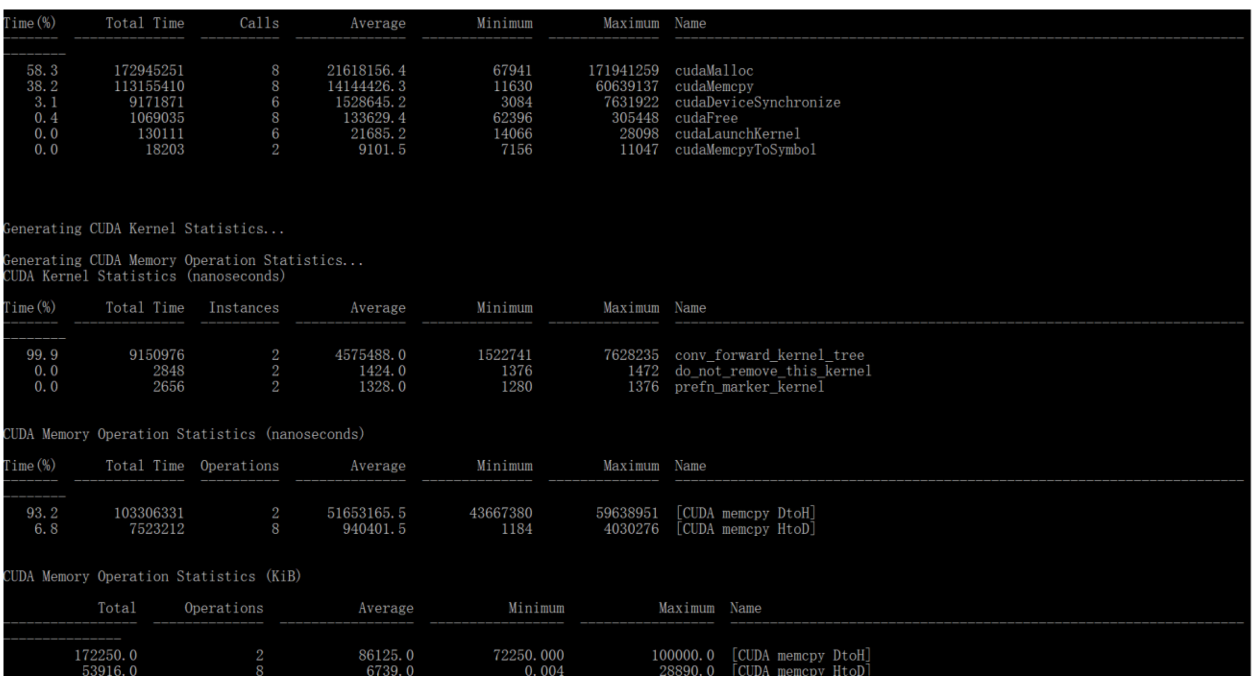
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.167445ms*	*0.769079ms*	*0m1.139s*	*0.86*
1000	*1.51407ms*	*7.57744ms*	*0m9.947s*	*0.886*
10000	*14.9917ms*	*68.1ms*	*1m36.298s*	*0.8714*

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

No. The profiling result shows that it is even a little worse than the baseline. That may be because we have use __syncthreads() and there may be a lot of control divergence, as we can see in the figure of the warp states.





Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
58.3	172945251	8	21618156.4	67941	171941259	cudaMalloc
38.2	113155410	8	14144426.3	11630	60639137	cudaMemcpy
3.1	9171871	6	1528645.2	3084	7631922	cudaDeviceSynchronize
0.4	1069035	8	133629.4	62396	305448	cudaFree
0.0	130111	6	21685.2	14066	28098	cudaLaunchKernel
0.0	18203	2	9101.5	7156	11047	cudaMemcpyToSymbol

Generating CUDA Kernel Statistics...

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

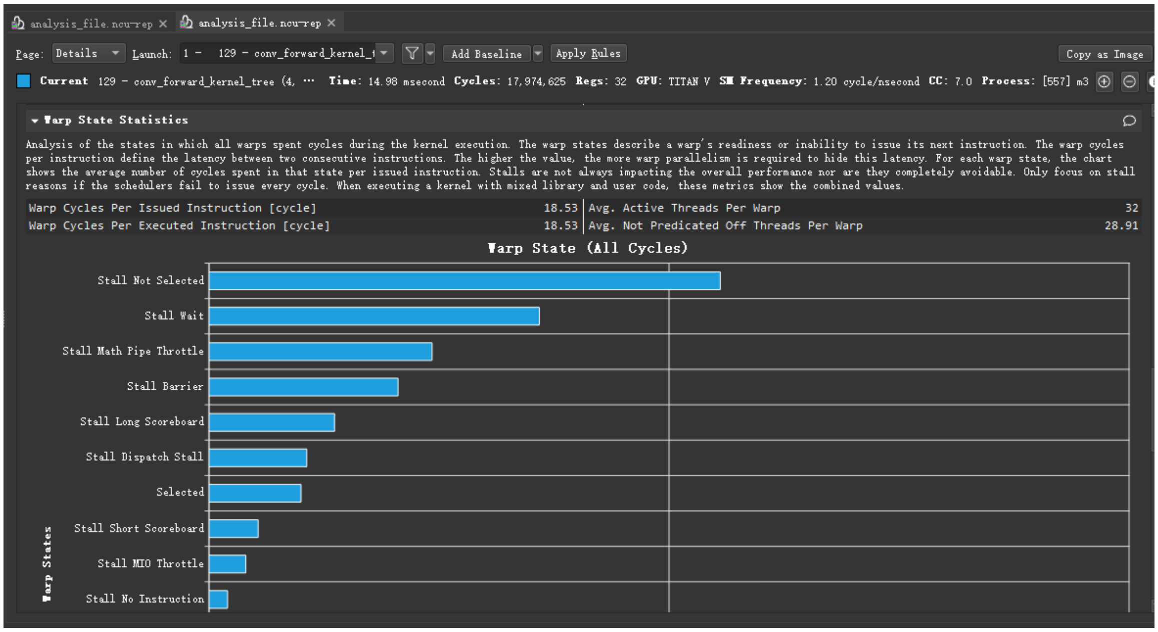
Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	9150976	2	4575488.0	1522741	7628235	conv_forward_kernel_tree
0.0	2848	2	1424.0	1376	1472	do_not_remove_this_kernel
0.0	2656	2	1328.0	1280	1376	prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
93.2	103306331	2	51653165.5	43667380	59638951	[CUDA memcpy DtoH]
6.8	7523212	8	940401.5	1184	4030276	[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]
53916.0	8	6739.0	0.004	28890.0	[CUDA memcpy HtoD]



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

Course slides and previous MP code.

6. **Optimization 6: *Multiple kernel implementations for different layer sizes (1 point)***

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

Multiple kernel implementations for different layer sizes. Because it is easy to implement, we only need to try and figure out which method is good for different layer sizes.

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?

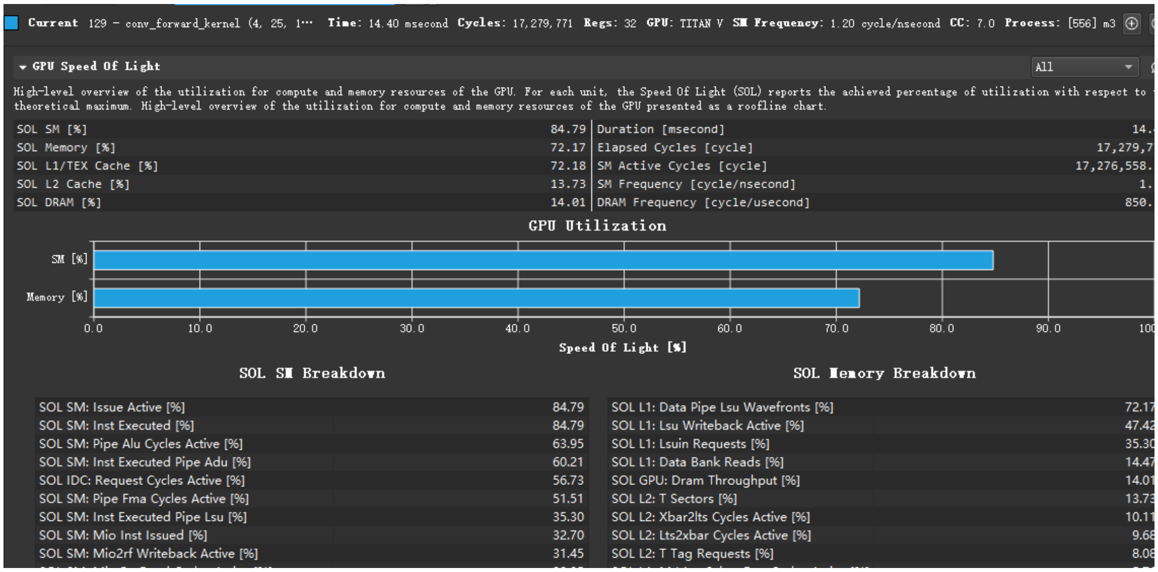
It just combines two methods together due to different layer sizes. I think it will improve the performance since we adjust according to different input parameters. It synergizes with tile shared convolution and weighted matrix in constant memory and reduction tree.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.11832ms*	*1.04847ms*	*0m4.221s*	*0.86*
1000	*2.54748ms*	*5.52425ms*	*0m12.218s*	*0.886*
10000	*17.3814ms*	*49.1319ms*	*1m40.545s*	*0.8714*

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

Yes. Because we try different methods according to different input parameters



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

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
59.6	170903618	8	21362952.2	64732	169922296	cudaMalloc
37.7	108201909	8	13525238.6	16851	58537672	cudaMemcpy
2.3	6713409	6	1118901.5	2916	4698529	cudaDeviceSynchronize
0.3	898373	8	112296.6	55200	231145	cudaFree
0.0	126563	6	21093.8	12742	27013	cudaLaunchKernel
0.0	16608	2	8304.0	6813	9795	cudaMemcpyToSymbol

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
99.9	6697713	2	3348856.5	2000370	4697343	conv_forward_kernel_shared
0.0	2656	2	1328.0	1280	1376	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
93.3	98854482	2	49427241.0	41268418	57586064	[CUDA memcpy DtoH]
6.7	7128623	8	891077.9	1216	3822854	[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]
53916.0	8	6739.0	0.004	28890.0	[CUDA memcpy HtoD]

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

7. **Optimization 7:** ***Shared memory matrix multiplication and input matrix unrolling (3 points)***

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

   Shared memory matrix multiplication and input matrix unrolling. Because it is mentioned in the lecture and it is interesting.

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?

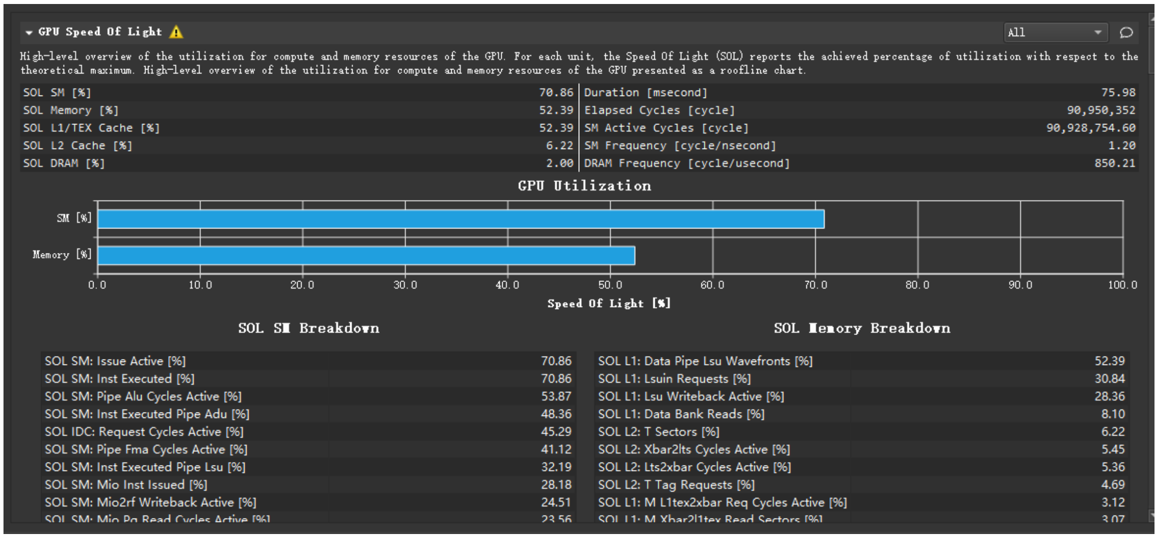
   It unrolls the matrix of input of each image and do matrix-matrix multiplication to calculate convolution. I do not know whether it will increase the performance since it is another baseline method. I also use tiled matrix to improve its performance. It synergizes with weight matrix in constant memory.

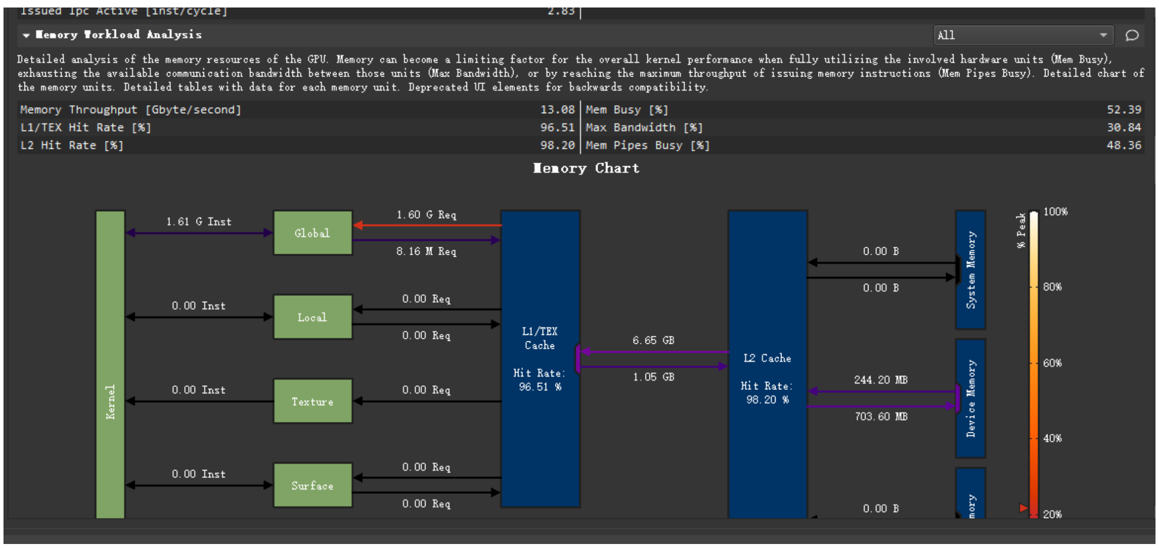
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.14645ms	0.75349ms	0m1.031s	0.86
1000	1.4175ms	8.7144ms	0m9.457s	0.886
10000	15.778ms	67.3ms	1m39.928s	0.8714

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

No, it is a little worse than normal convolution baseline. Because the matrix-matrix multiplication has no obvious advantage if no advanced optimization added.





e. What references did you use when implementing this technique?
Course slides and textbook.