Name: *Joseph Thomas*
NetID: *jpt7*
Section: *AL2*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai_amd64_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.198917ms*	*0.585584ms*	*0m2.964s*	*0.86*
1000	*2.77284ms*	*6.33284ms*	*0m12.430s*	*0.886*
5000	*14.748ms*	*32.8914ms*	*0m51.459s*	*0.871*

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

Time(%)      Total Time       Calls          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   69.1       576325657          20      28816282.9           36210       285361046  cudaMemcpy
   23.8       198959200          20       9947960.0            2913       194824399  cudaMalloc
    4.9        40668641          10       4066864.1            4007        32019231  cudaDeviceSynchronize
    1.9        15810312          10       1581031.2           19018        15562997  cudaLaunchKernel
    0.3         2810686          20        140534.3            2102          580987  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        40643429           6       6773904.8            7360        32017498  conv_forward_kernel
    0.0            2816           2          1408.0            1376            1440  do_not_remove_this_kernel
    0.0            2592           2          1296.0            1280            1312  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   91.7       512753764           6      85458960.7           23328       284642563  [CUDA memcpy DtoH]
    8.3        46333124          14       3309508.9            1152        24003756  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name

-----------------  --------------  -----------------  -----------------  -----------------  --------------------
----
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]

         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

- **Optimization 1: Weight matrix (kernel values) in constant memory (3 point)**

1.

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

I chose to implement weight matrix (kernel values) in constant memory because this technique allows a quicker access to constant memory comparing to accessing from global memory.

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 implementation works by making constant memory copy the host mask and make it readily available for blocks to access rather than blocks accessing global memory like it used to in M2 implementation. I think the optimization would work assuming that accessing constant memory is quicker than accessing global memory. No, this optimization doesn't synergizes with any other optimization.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.172613 ms*	*0.587303 ms*	*0m1.669s*	*0.86*
1000	*1.56391 ms*	*5.70719 ms*	*0m10.278s*	*0.886*
5000	*7.70291 ms*	*28.9042 ms*	*49.097s*	*0.871*

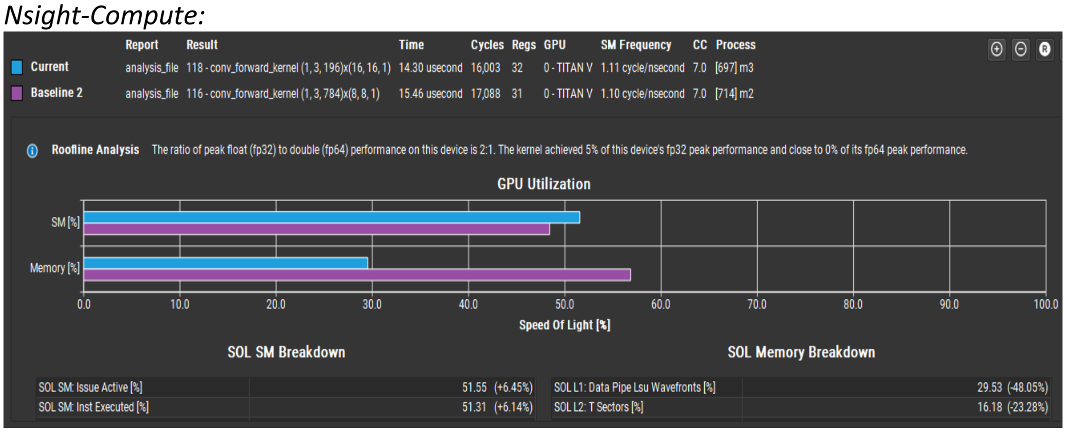
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, it was successful because it is quicker to access constant memory than global memory. While comparing the nsys of M2 and optimization, it is observable that cudaMemcpy and cudaMemcpyToSymbol combined time of the optimization is comparably below cudaMemcpy time of M2. It is also observable that this difference has also effect on kernel time, giving assurance that the optimization is successful.

Nsys:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   68.5       541865144          20      27093257.2           26228       290676856  cudaMemcpy
   24.5       193600378          20       9680018.9            2835       190439130  cudaMalloc
    4.7        36936474          10       3693647.4            3957        29051955  cudaDeviceSynchronize
    2.1        16351220          10       1635122.0            9601        16201442  cudaLaunchKernel
    0.3         2452784          20        122639.2            3615          627378  cudaFree
    0.0          137031           6         22838.5           18386           25909  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
  100.0        36911984           6       6151997.3            6944        29051025  conv_forward_kernel
    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kernel
    0.0            2528           2          1264.0            1216            1312  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   91.4       489429003           6      81571500.5           23200       289901555  [CUDA memcpy DtoH]
    8.6        45895432          14       3278245.1            1152        23991995  [CUDA memcpy HtoD]
    0.0           10944           6          1824.0            1536            2240  [CUDA memcpy DtoD]
CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name

-----------------  --------------  -----------------  -----------------  -----------------  ------------------
----
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]

         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]

             14.0               6                2.4              0.316               12.0  [CUDA memcpy DtoD]
```

Nsight-Compute:



e. What references did you use when implementing this technique?
Lecture slides and Textbook

2. **Optimization 2: Input channel reduction: tree**
   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      I chose Input channel reduction: tree as my optimization because input reduction tree increase parallelism.

   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?
      Optimization works by using tree reduction to increase parallelism by assigning each thread to a channel and then making use of reduction method to achieve higher parallelism. I think the optimization would increase performance of the forward convolution because it manages many channels at same time.

   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.205646 ms*	*0.678121 ms*	*0m1.581s*	*0.86*
1000	*1.89966 ms*	*6.63198 ms*	*0m10.352s*	*0.886*
5000	*9.42865 ms*	*33.073 ms*	*0m51.558s*	*0.871*

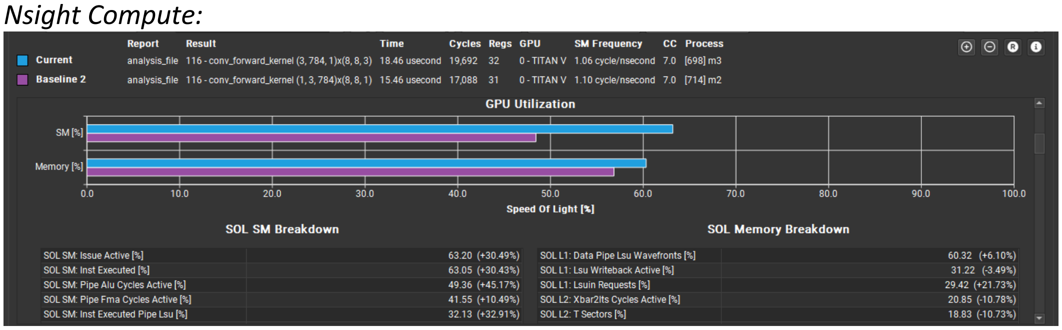
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

      No, it was not successful and it is comparably worse than M2 baseline. I think the reason could be because of the use of syncthreads, which is observable when you compare the difference in kernel time shown in nsys.

Nsys:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   70.2       557524386          20      27876219.3           32290       299894313  cudaMemcpy
   21.6       171830843          20       8591542.2            2890       168125335  cudaMalloc
    5.4        42560390          10       4256039.0            4072        33113590  cudaDeviceSynchronize
    2.5        19512742          10       1951274.2           15772        19287799  cudaLaunchKernel
    0.3         2701858          20        135092.9            2356          600550  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        42529424           6       7088237.3            6784        33111508  conv_forward_kernel
    0.0            9280           2          4640.0            1312            7968  prefn_marker_kernel
    0.0            2880           2          1440.0            1408            1472  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   92.7       499937365           6      83322894.2           12640       299180270  [CUDA memcpy DtoH]
    7.3        39633990          14       2830999.3            1152        20964875  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
----
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

Nsight Compute:



e. What references did you use when implementing this technique?
Lecture slides and Textbook

3. **Optimization 3: Multiple kernel implementations for different layer sizes**
   (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.

      I chose Multiple kernel implementation as my optimization because I think that kernel incorporating different layer sizes will improve 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?

      It has two separate kernels with the same algorithm but optimized to incorporate different layer sizes. I think that this optimization would increase performance of the forward convolution because this implementation modifies its parameters to incorporate unique inputs. This optimization synergizes with Weight matrix (kernel values) in constant memory.

   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.176561 ms*	*0.586961 ms*	*0m4.156s*	*0.86*
1000	*1.5644 ms*	*5.70981 ms*	*0m11.248s*	*0.886*
5000	*7.7109 ms*	*28.9908 ms*	*0m52.801s*	*0.871*

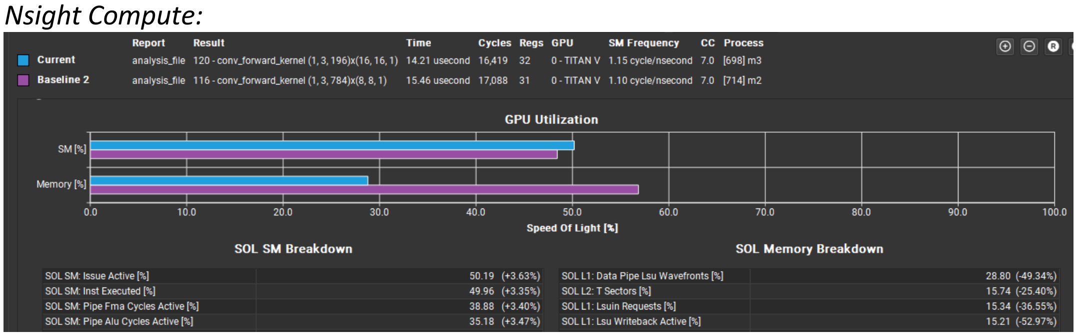
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

      Yes, implementing this optimization was successful in improving performance because we successfully incorporated parameters to match with specific unique inputs. It is observable in nsys that the kernel time has a significant improvement when compared to M2 baseline and Weight matrix (kernel values) in constant memory optimization which was synergized with this optimization.

Nsys:

```
CUDA Memory Operation Statistics (KiB)

           Total      Operations             Average             Minimum             Maximum  Name

----------------  -------------  ----------------  ----------------  ----------------  -------------------
----
        862672.0              6          143778.7           148.535          500000.0  [CUDA memcpy DtoH]

        276206.0             14           19729.0             0.004          144453.0  [CUDA memcpy HtoD]

            14.0              6               2.4             0.316              12.0  [CUDA memcpy DtoD]

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   70.0       550587195          20      27529359.7           28116       304621197  cudaMemcpy
   22.7       178988056          20       8949402.8            2598       175356525  cudaMalloc
    4.7        36800454          10       3680045.4            3497        28966368  cudaDeviceSynchronize
    2.2        17638187          10       1763818.7           10117        17478125  cudaLaunchKernel
    0.3         2700026          20        135001.3            3406          656955  cudaFree
    0.0          162344           6         27057.3           24230           29098  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        36779600           6       6129933.3            7040        28965795  conv_forward_kernel
    0.0            2912           2          1456.0            1440            1472  do_not_remove_this_kernel
    0.0            2560           2          1280.0            1248            1312  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   92.9       505565812           6      84260968.7           12736       303842333  [CUDA memcpy DtoH]
    7.1        38493059          14       2749504.2            1152        19631423  [CUDA memcpy HtoD]
    0.0           11136           6          1856.0            1376            2272  [CUDA memcpy DtoD]
```

Nsight Compute:



e. What references did you use when implementing this technique?
Lecture slides and Textbook

4. **Optimization 4: Tuning with restrict and loop unrolling (3 points)**

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

I chose to implement Tuning with restrict and loop unrolling as my optimization because restrict and unrolling improves performance by implementing a single iteration rather than multiple iteration.

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?

In the implementation every data pointer is changed to restricted pointers. A loop is then introduced to iterate through elements of the matrices to allow an advantage to small K values. I think that this optimization would increase performance of the forward convolution because restrict and unrolling improves performance by implementing a single iteration rather than multiple iteration. This optimization don't synergize with any of the other previous optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.215549 ms*	*0.572844 ms*	*0m1.777s*	*0.86*
1000	*2.02565 ms*	*5.59096 ms*	*0m10.812s*	*0.886*
5000	*10.036 ms*	*27.8985 ms*	*0m51.108s*	*0.871*

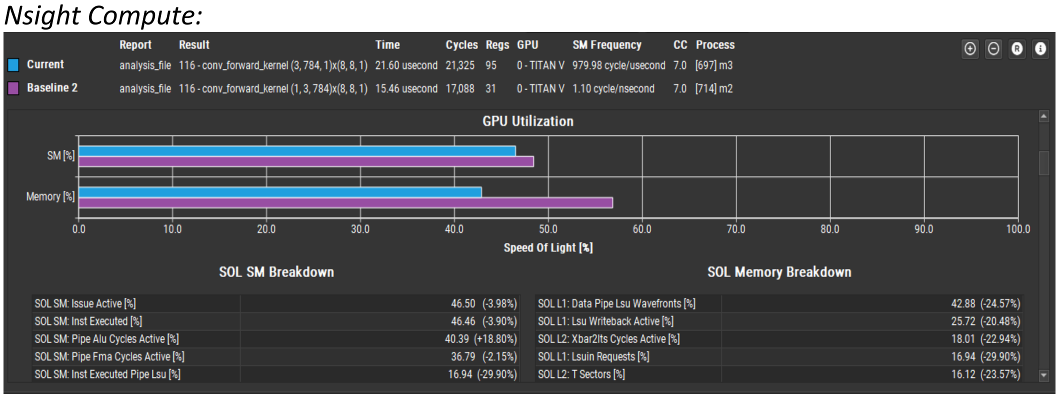
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization was successful in improving performance because restrict and unrolling has increased performance by implementing a single iteration rather than multiple iteration. It is observable in nsys that both cudaMemcpy and cudaMalloc times are significantly lower while comparing to baseline M2. Which is also reflected in the difference in kernel execution time. It is also observable in the Nsight compute that there is a major difference in Memory(%) indicating it is running efficiently than M2 baseline.

Nsys:

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

Time(%)    Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   70.2       544964030          20      27248201.5           28501       295680013  cudaMemcpy
   22.4       173957519          20       8697875.9            2824       170617523  cudaMalloc
    4.9        38317121          16       2394820.1             812        27930975  cudaDeviceSynchronize
    2.2        16992043          10       1699204.3           15553        16790758  cudaLaunchKernel
    0.3         2323208          20        116160.4            2758          569580  cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)    Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
  100.0        38172809           6       6362134.8            7008        27928627  conv_forward_kernel
    0.0            2816           2          1408.0            1376            1440  do_not_remove_this_kernel
    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)    Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   92.5       498370400           6      83061733.3           12608       294933363  [CUDA memcpy DtoH]
    7.5        40201947          14       2871567.6            1152        20478155  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name

----------------  -------------  -----------------  -----------------  -----------------  ------------------
----
         862672.0              6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]

         276206.0             14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

Nsight Compute:



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

5. **Optimization 5:** Tiled shared memory convolution (2 points)

(Delete this section if you did not implement this many optimizations.)

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

   I chose Tiled shared memory convolution as my optimization technique because using shared memory will reduce the time consumed for accessing input features.

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 improve memory reuse rather than accessing global memory which tends to be slow. I think that this optimization would increase performance of the forward convolution, because it is quicker to access shared memory than global memory. This optimization doesn't synergize with any other previous optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.751419 ms*	*1.19764 ms*	*0m2.830s*	*0.86*
1000	*2.54745 ms*	*6.93911 ms*	*0m11.514s*	*0.886*
5000	*11.7706 ms*	*32.3134 ms*	*0m52.301s*	*0.871*

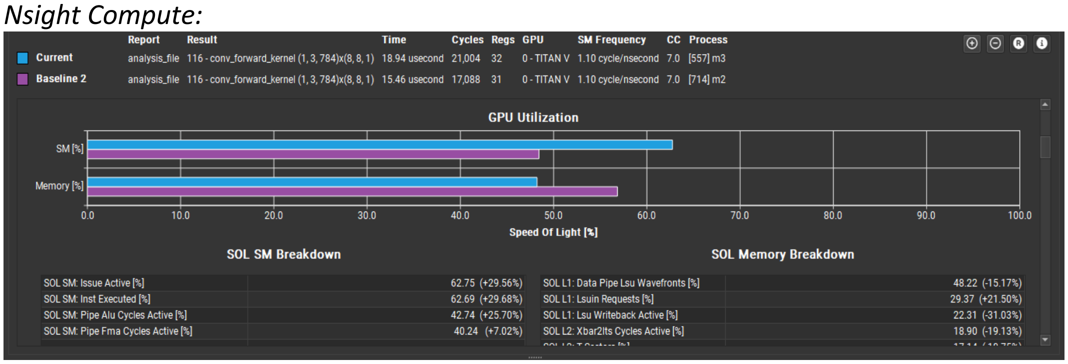
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

   No, the optimization was not successful in improving performance because there was not much reuse of the shared memory, which would have led to the misuse of resources in shared memory. It is also observable with increased kernel time while comparing it with M2 baseline. While it is evident also in Nsight Compute with the difference in SM(%) while comparing with M2 baseline.

Nsys:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   64.6      1926927985          20      96346399.3            2416      1921392706  cudaMalloc
   19.3       576558759          20      28827937.9           29757       308973112  cudaMemcpy
   14.1       419920123          20      20996006.1            2575       236940449  cudaFree
    1.5        44363512          10       4436351.2            6336        31827333  cudaDeviceSynchronize
    0.5        15830224          10       1583022.4           23968        15583101  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        43134822           6       7189137.0           11936        31823519  conv_forward_kernel
    0.0            2944           2          1472.0            1440            1504  do_not_remove_this_kernel
    0.0            2752           2          1376.0            1312            1440  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   91.8       525092701           6      87515450.2           23232       308200198  [CUDA memcpy DtoH]
    8.2        46616500          14       3329750.0            1120        24690146  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name

-----------------  --------------  -----------------  -----------------  -----------------  ------------------
----
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]

         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

Nsight Compute:



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

6. **Optimization 6:** Sweeping various parameters to find best (0.5 point)
(Delete this section if you did not implement this many optimizations.)

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

I chose Sweeping various parameters as my optimization because it needs very little changes to the code and can result to improvement in 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?

For this optimization I tested Tile Sizes and fixed on Tile Size of 24. I think that the optimization would increase performance of the forward convolution because trying different Tile Size can result to a better performance. This optimization doesn't synergize with any of the other optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.243166 ms*	*0.687001 ms*	*0m1.658s*	*0.86*
1000	*2.29288 ms*	*6.77237 ms*	*0m10.609s*	*0.886*
5000	*11.337 ms*	*34.2141 ms*	*0m51.766s*	*0.871*

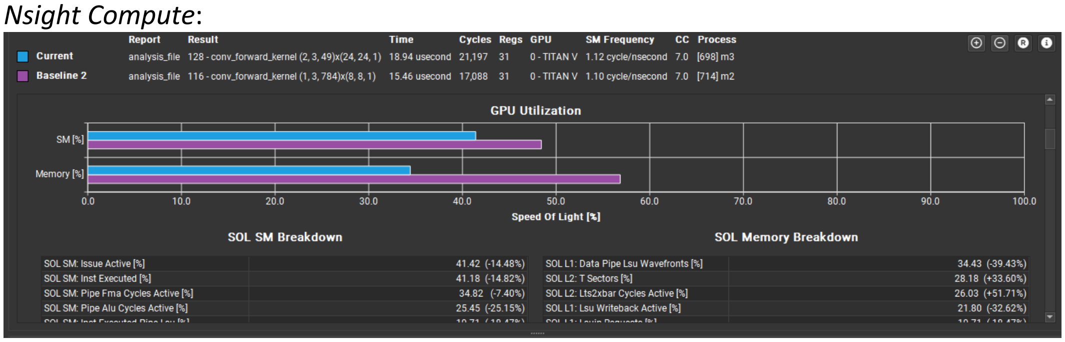
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Implementing this optimization is successful in improving performance because as seen with Op time which has decreased significantly from M2 baseline. It is also observable in nsys where the kernel time has improved significantly show that the performance has improved. It is also represented in Nsight Compute where both SM(%) and Memory(%) are lower compared to M2 baseline.

Nsys:

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   68.9       532208961          20      26610448.1           35631       284599990  cudaMemcpy
   22.7       175285321          20       8764266.1            3400       172401833  cudaMalloc
    5.9        45439970          10       4543997.0            3364        33885381  cudaDeviceSynchronize
    2.1        16397659          10       1639765.9           25356        16144342  cudaLaunchKernel
    0.4         2751679          20        137584.0            2706          616860  cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        45414731           6       7569121.8            8352        33883721  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
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   91.3       482445604           6      80407600.7           23232       283893816  [CUDA memcpy DtoH]
    8.7        45906150          14       3279010.7            1152        23995517  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations              Average            Minimum              Maximum  Name

-----------------  --------------  -----------------  -----------------  -----------------  ------------------
----
         862672.0               6           143778.7            148.535             500000.0  [CUDA memcpy DtoH]

         276206.0              14            19729.0              0.004             144453.0  [CUDA memcpy HtoD]
```

Nsight Compute:



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