Name: *John Li*

NetID: *Johnwl2*

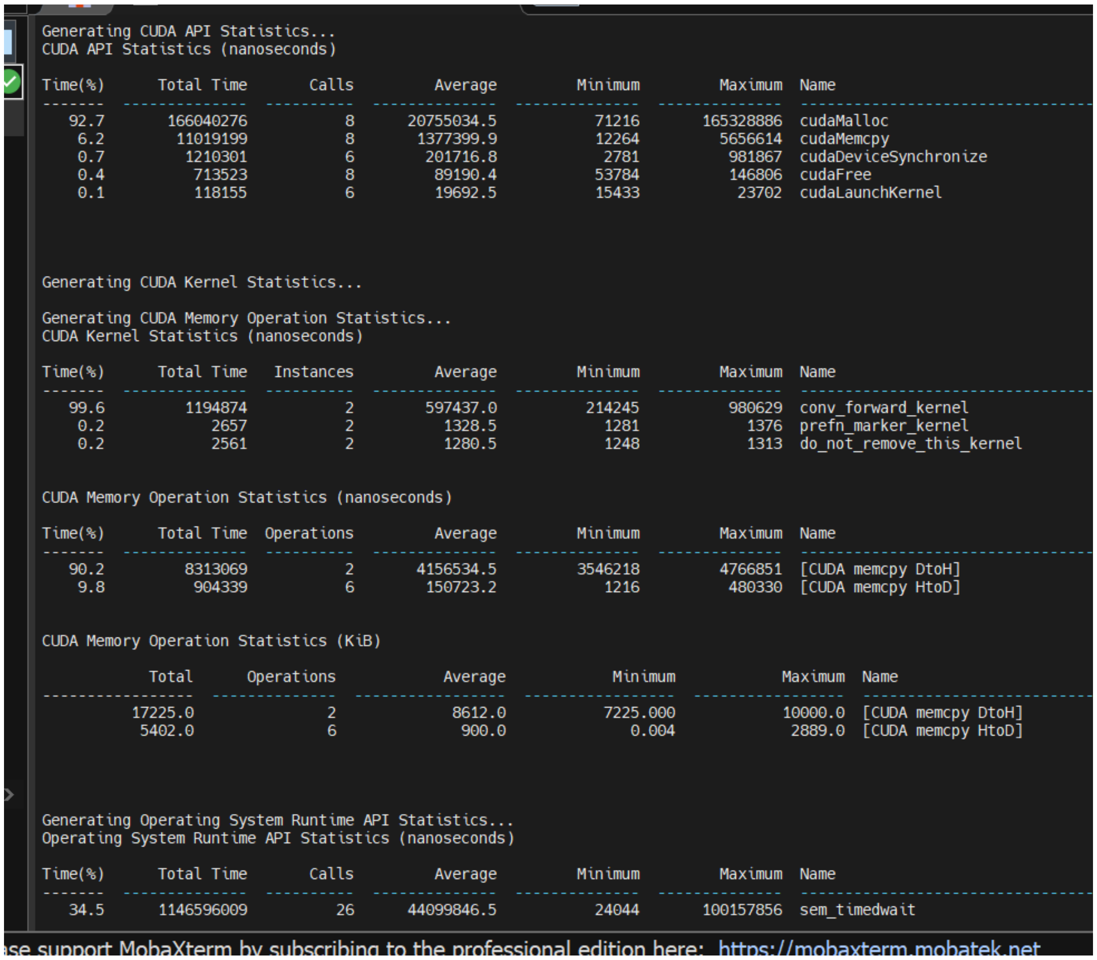
Section: *AB*

ECE 408/CS483 Milestone 3 Report

1. 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 for this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.178091 ms*	*0.636865 ms*	*0m1.730s*	*0.86*
1000	*2.21351 ms*	*8.81894 ms*	*0m11.151s*	*0.886*
10000	*21.6647 ms*	*86.24 ms*	*1m53.180s*	*0.8714*

Nsys profiling results for batch size 100 below:



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

Time(%)      Total Time       Calls          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------------
   92.7       166040276           8      20755034.5           71216       165328886  cudaMalloc
    6.2        11019199           8       1377399.9           12264         5656614  cudaMemcpy
    0.7         1210301           6        201716.8            2781          981867  cudaDeviceSynchronize
    0.4          713523           8         89190.4           53784          146806  cudaFree
    0.1          118155           6         19692.5           15433           23702  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------------
   99.6         1194874           2         597437.0          214245          980629  conv_forward_kernel
    0.2            2657           2           1328.5            1281            1376  prefn_marker_kernel
    0.2            2561           2           1280.5            1248            1313  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------------
   90.2         8313069           2        4156534.5         3546218         4766851  [CUDA memcpy DtoH]
    9.8          904339           6         150723.2            1216          480330  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ----------------------------
          17225.0               2             8612.0           7225.000            10000.0  [CUDA memcpy DtoH]
           5402.0               6              900.0              0.004             2889.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------------
   34.5      1146596009          26       44099846.5           24044       100157856  sem_timedwait
ase support MobaXterm by subscribing to the professional edition here:  https://mobaxterm.mobatek.net
```

2. **Optimization 1: *Sweeping various parameters to find best values***

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

      My first optimization to kick off from the baseline implementation is to sweep various parameters to find the most optimal values. I chose this optimization because I was curious as to how simply adjusting the values and sizes of certain variables and macros would impact the performance and efficiency of my convolution forward kernel.

   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 works by trying various sizes or values for the parameters in my forward convolution kernel (in this case, I found my TILE_WIDTH to have the most effective influence) and then determining which value is the best. I found TILE_WIDTH 12 to be the best. I think this optimization increases the performance of the forward convolution because a TILE_WIDTH of 12 is not too large and not too small to use for tiling. (My original TILE_WIDTH for the baseline was size 32.) Other values for TILE_WIDTH I tried were 4, 64 (TILE_WIDTH 64 didn't even give the correct accuracy), and 16, but their resulting performances weren't as strong compared to the performance when TILE_WIDTH = 12. This was my first optimization, so there were no optimizations prior to it: just the baseline before.

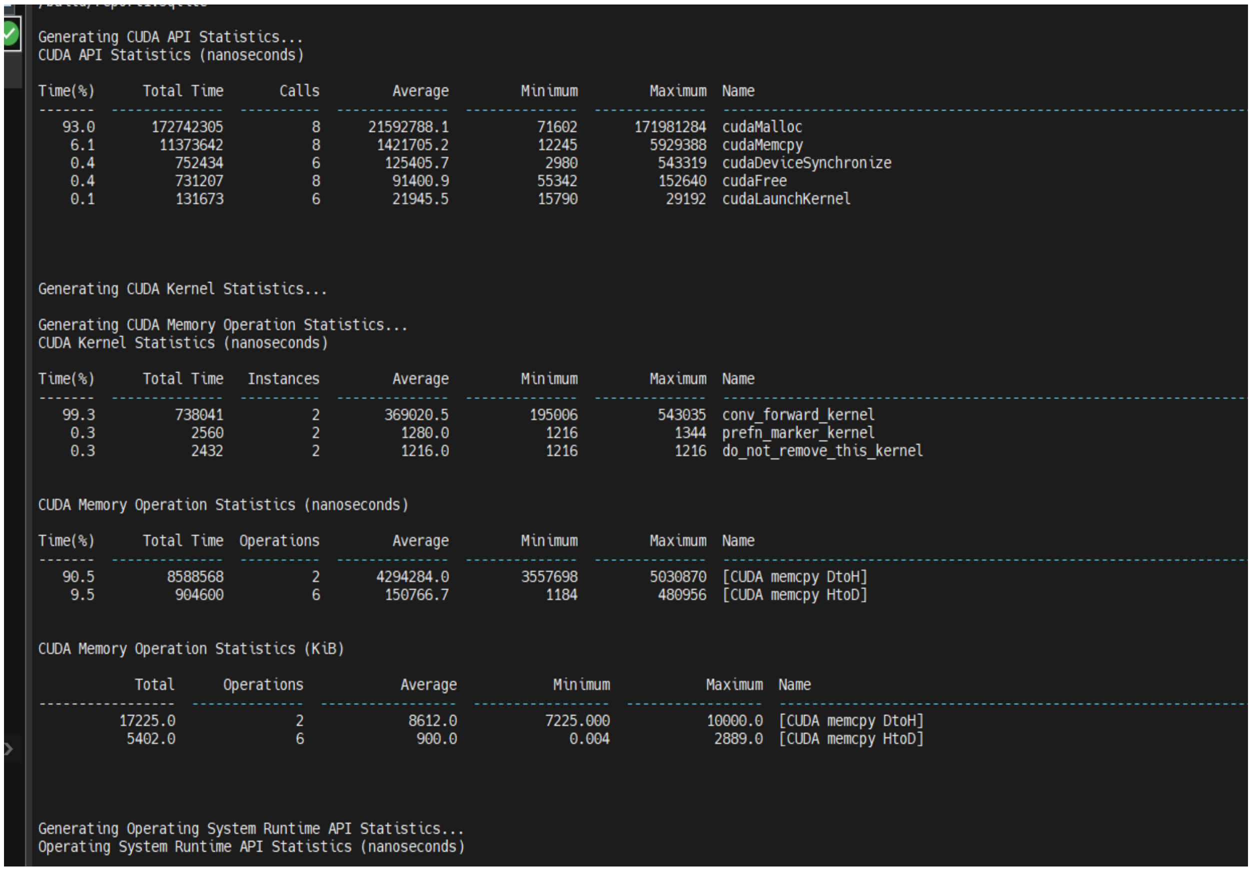
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.205364 ms*	*0.550425 ms*	*0m1.220s*	*0.86*
1000	*1.96903 ms*	*5.47458 ms*	*0m10.966s*	*0.886*
10000	*19.613 ms*	*54.4653 ms*	*1m46.744s*	*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, it was successful in improving performance. This is most likely because a tile size of 12 is not too large yet not too small for a kernel to perform forward convolution, so it ends up being the most ideal size. An alternative way to phrase this is that generically speaking, every dataset has an ideal size. And in my particular situation, the ideal tile size/width was 12.

Nsys profiling results for batch size 100 below:



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

Time(%)      Total Time       Calls          Average         Minimum         Maximum  Name
-------   --------------   ----------   --------------  --------------  --------------  ------------------------------------------------------------------------------------
   93.0        172742305            8       21592788.1           71602       171981284  cudaMalloc
    6.1         11373642            8        1421705.2           12245         5929388  cudaMemcpy
    0.4           752434            6         125405.7            2980          543319  cudaDeviceSynchronize
    0.4           731207            8          91400.9           55342          152640  cudaFree
    0.1           131673            6          21945.5           15790           29192  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average         Minimum         Maximum  Name
-------   --------------   ----------   --------------  --------------  --------------  ------------------------------------------------------------------------------------
   99.3           738041            2         369020.5          195006          543035  conv_forward_kernel
    0.3             2560            2           1280.0            1216            1344  prefn_marker_kernel
    0.3             2432            2           1216.0            1216            1216  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average         Minimum         Maximum  Name
-------   --------------   ----------   --------------  --------------  --------------  ------------------------------------------------------------------------------------
   90.5          8588568            2        4294284.0         3557698         5030870  [CUDA memcpy DtoH]
    9.5           904600            6         150766.7            1184          480956  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------   --------------   -----------------  -----------------  -----------------  ------------------------------------------------------------------------
          17225.0                2             8612.0           7225.000            10000.0  [CUDA memcpy DtoH]
           5402.0                6              900.0              0.004             2889.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)
```

As you can see from the nsys profiling, the average time it took for the forward convolution kernel implementation is 369,020.5 nanoseconds which is faster than the baseline forward convolution kernel average time of 597,437 nanoseconds. The same trend is present for other time statistics like the total time, minimum time, etc. in the profiling. Furthermore, the OP times and total execution times under this implementation are faster than those of the baseline. For example the OP times for this optimization for a batch size of 10000 was ~19.6 ms and ~54.5 ms respectively, while the baseline implementation had times of 21.67 ms and 86.24 ms for the same batch size.

e. What references did you use when implementing this technique?
   None.

3. **Optimization 2: *Sweeping various parameters to find best values* + Tuning with restrict and loop unrolling because restricting**

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

      I chose to stack my previous optimization with the "Tuning with restrict and loop unrolling" because I believed tuning certain parameters with the restrict keyword and unrolling the loops in my convolution forward kernel manually would be faster than structuring it as regular for loops. Furthermore, this optimization is simple to implement and understand.

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?

   First you tweak certain parameters passed into the convolution forward kernel function with the restrict keyword. In particular, you change const float * x and const float * k to const float * __restrict__ x and const float * __restrict__ k. Then you need to take the double for loop which originally looped from 0 to K twice (because it was a x2 for loop) and "unroll" it or manually perform the floating-point add computation with the actual numbers 0-6 (7 times because kernel size is 7x7). I believe this optimization would increase the performance of the forward convolution because having the kernel go through a loop is slightly less efficient than calculating the computation directly multiple times. Furthermore, I believe using the __restrict__ keyword will significantly help the compiler optimize code because it doesn't have to worry that a write to a restrict pointer will cause a value read from another restrict pointer to change. Also, this optimization synergizes with the previous "sweeping various parameters to find best values" optimization; I stacked this current optimization on top of the previous one.

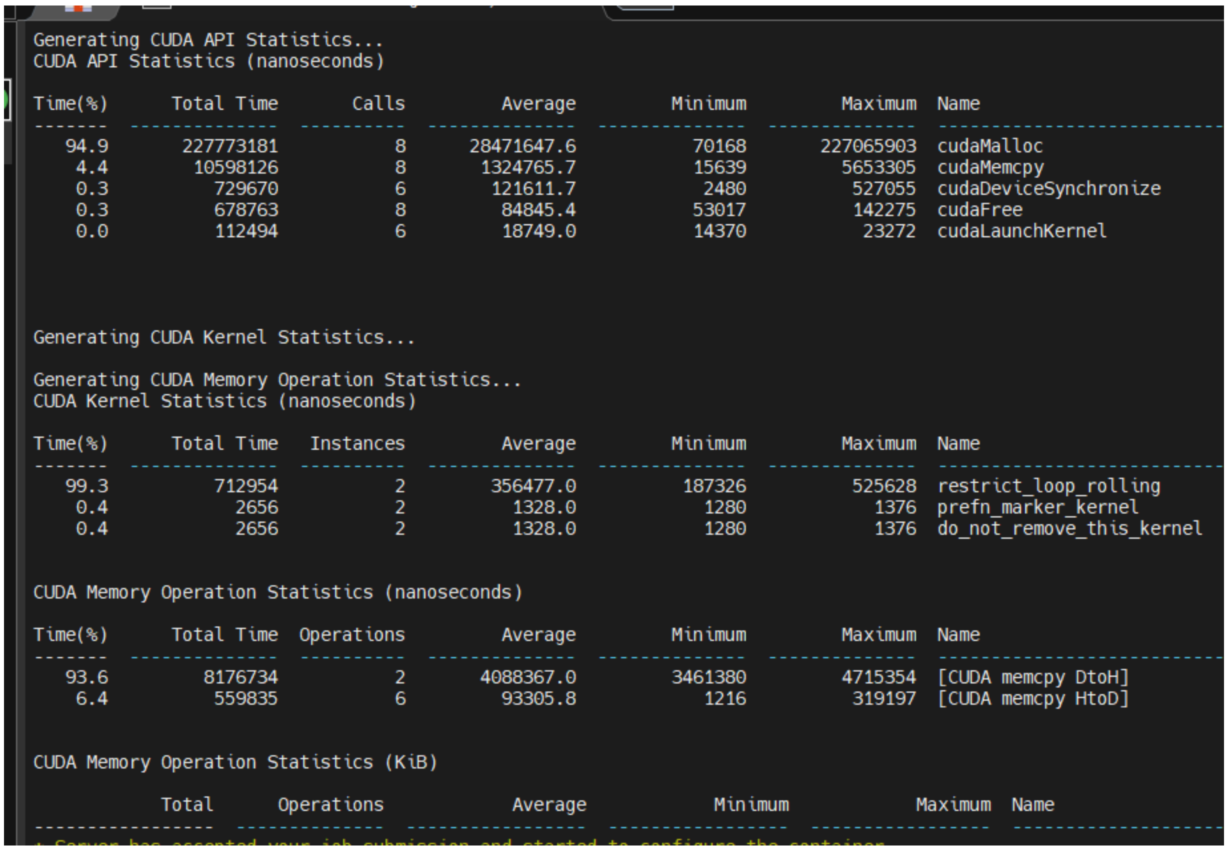
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.197431 ms*	*0.536113 ms*	*0m1.168s*	*0.86*
1000	*1.98221 ms*	*5.48225 ms*	*0m10.395s*	*0.886*
10000	*20.3537 ms*	*54.8231 ms*	*1m44.959s*	*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 was slightly successful in improving performance. First, the __restrict__ keyword makes the compiler not have to worry that a write to a restrict pointer will cause a value read from another restrict pointer to change. As for the unrolling aspect, it is useful because it eliminates the conditional check and branch portion of a loop, which can be expensive. It also allows the compiler to pre-compute what the indices and values are and insert them in place where the loop counter would be. The biggest setback is that it can bloat code size which can affect memory usage. Since I practically copy-pasted the same line/computation 49 times, that appears to bloat my code size which may offset some of the headway I've been making in regards to performance improvements. Thus, the overall resulting performance is only slightly more successful as compared to before.

Nsys profiling results for batch size 100 below:



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

Time(%)      Total Time       Calls          Average          Minimum          Maximum  Name
-------  --------------  ----------  ---------------  ---------------  ---------------  ------------------------
   94.9       227773181           8       28471647.6            70168        227065903  cudaMalloc
    4.4        10598126           8        1324765.7            15639          5653305  cudaMemcpy
    0.3          729670           6         121611.7             2480           527055  cudaDeviceSynchronize
    0.3          678763           8          84845.4            53017           142275  cudaFree
    0.0          112494           6          18749.0            14370            23272  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average          Minimum          Maximum  Name
-------  --------------  ----------  ---------------  ---------------  ---------------  ------------------------
   99.3          712954           2         356477.0           187326           525628  restrict_loop_rolling
    0.4            2656           2           1328.0             1280             1376  prefn_marker_kernel
    0.4            2656           2           1328.0             1280             1376  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average          Minimum          Maximum  Name
-------  --------------  ----------  ---------------  ---------------  ---------------  ------------------------
   93.6         8176734           2        4088367.0          3461380          4715354  [CUDA memcpy DtoH]
    6.4          559835           6          93305.8             1216           319197  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
```

From the Nsys profiling results above, unrolling + leveraging *__restrict__* cut down slightly on overall times compared to the previous optimization where I sweeped various parameters to find the most optimal values (i.e. the TILE_WIDTH being 12). For example, the average time is 356,477 nanoseconds and the total time is 712954 ns for this optimization, which is slightly faster than the total and average times of the prior optimization which were 738041 ns for total time and 369020 ns for average time. Furthermore, the OP times and total execution times under this implementation are, in general, slightly faster or practically the same than those of the previous implementation. For instance, for a batch size of 100, the respective OP times were 0.197431 ms and 0.536113 ms, while the previous optimization's OP times (without this current one) for the same batch size were 0.205364 ms and

0.550425 ms for the same batch size.

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

   https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

4. **Optimization 3: *Weight Matrix (kernel values) in constant memory + Tuning with restrict keywords + loop unrolling + Sweeping various parameters to find best values***
   (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?

      Weight matrix (kernel values) in constant memory and the baseline GPU convolution forward kernel (from PM2). I chose to implement this particular optimization technique because I believed using (extra) constant memory to perform the primary computations for the GPU convolution forward kernel (PM2) would make the computations as a whole faster.

   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?

      *First you declare a block of constant memory globally with size 3136. 3136 comes from kernel dimension * kernel dimension * sizeof(float) * C (# of channels) = 7 * 7 * 4 * 16. Then you insert or replace the block of constant memory into the forward convolution algorithm/implementation. As long as threads in the same warp access the same address, leveraging constant memory is a very fast and effective solution. In my case, all the threads I am using in each warp access the same address. This optimization synergizes with "tuning with restrict and loop unrolling" and "sweeping various parameters to find best values"; in fact, I stacked this optimization on top of those previous ones.*

   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).

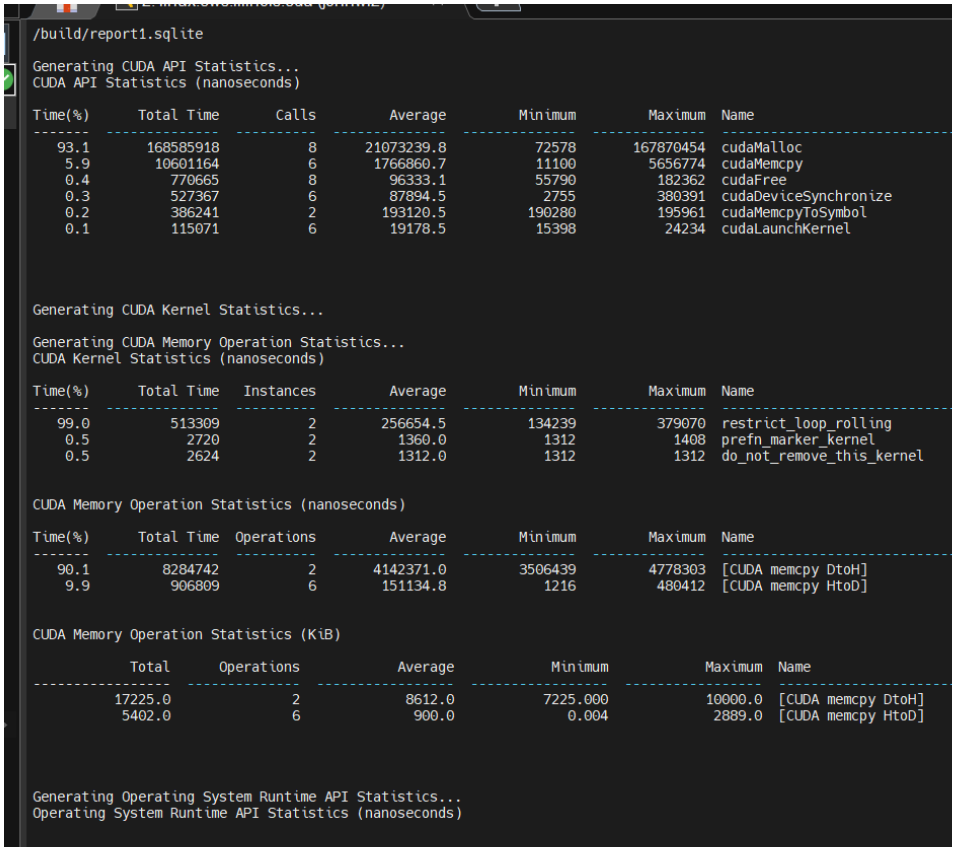
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.143663 ms*	*0.389186 ms*	*1.143 s*	*0.86*

1000	*1.69953 ms*	*3.93012 ms*	*10.189 s*	*0.886*
10000	*16.8401 ms*	*39.1409 ms*	*1 min 42.878 s*	*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, this optimization was successful in improving performance because as long as threads in the same warp access the same address (which is precisely what occurred in my program), leveraging constant memory becomes a very fast and effective solution.

Nsys profiling results for batch size 100 below:



```
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   93.1       168585918           8      21073239.8           72578       167870454  cudaMalloc
    5.9        10601164           6       1766860.7           11100         5656774  cudaMemcpy
    0.4          770665           8         96333.1           55790          182362  cudaFree
    0.3          527367           6         87894.5            2755          380391  cudaDeviceSynchronize
    0.2          386241           2        193120.5          190280          195961  cudaMemcpyToSymbol
    0.1          115071           6         19178.5           15398           24234  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   99.0          513309           2        256654.5          134239          379070  restrict_loop_rolling
    0.5            2720           2          1360.0            1312            1408  prefn_marker_kernel
    0.5            2624           2          1312.0            1312            1312  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   90.1         8284742           2       4142371.0         3506439         4778303  [CUDA memcpy DtoH]
    9.9          906809           6        151134.8            1216          480412  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations           Average           Minimum           Maximum  Name
-----------------  --------------  ----------------  ----------------  ----------------  ----------------------------
          17225.0               2            8612.0          7225.000           10000.0  [CUDA memcpy DtoH]
           5402.0               6             900.0             0.004            2889.0  [CUDA memcpy HtoD]



Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)
```

As one can see from the Nsys profiling results above, the average time is 256655.5 ns and the total time is 513309 ns. This is significantly better than the previous runtimes, which were 738041 ns for total time and 369020 ns for average time. The same trend can be extrapolated toward the other time metrics (min time, etc.) as well. Moreover, the OP times and total execution times under this implementation are faster than those of the previous implementation (first and second OP times were 16.8401 ms and 39.1409 ms for a batch size of 10000). Leveraging constant memory, along with having a TILE_WIDTH = 12 (as a result of sweeping for the most optimal parameters) and tuning with __restrict__ and loop unrolling, appears to be my best optimization yet.

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

http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html

5. **Optimization 4: Tiled shared memory convolution + sweeping various parameters to find the best values.**

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

I chose to do tiled shared memory convolution because I believed utilizing shared memory and more advanced tiling methods would cut down on overall runtimes since the fundamental forward convolution kernel don't use such strategies to their advantage.

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?

At its core, this optimization uses shared memory to assist with forward convolution, but the general forward convolution algorithm would remain the same (so same idea as baseline implementation). So I think this optimization would enhance performance because it makes use of extra (shared) memory. Also, I synergized this optimization with the first optimization: sweeping various parameters to find the best values.

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).

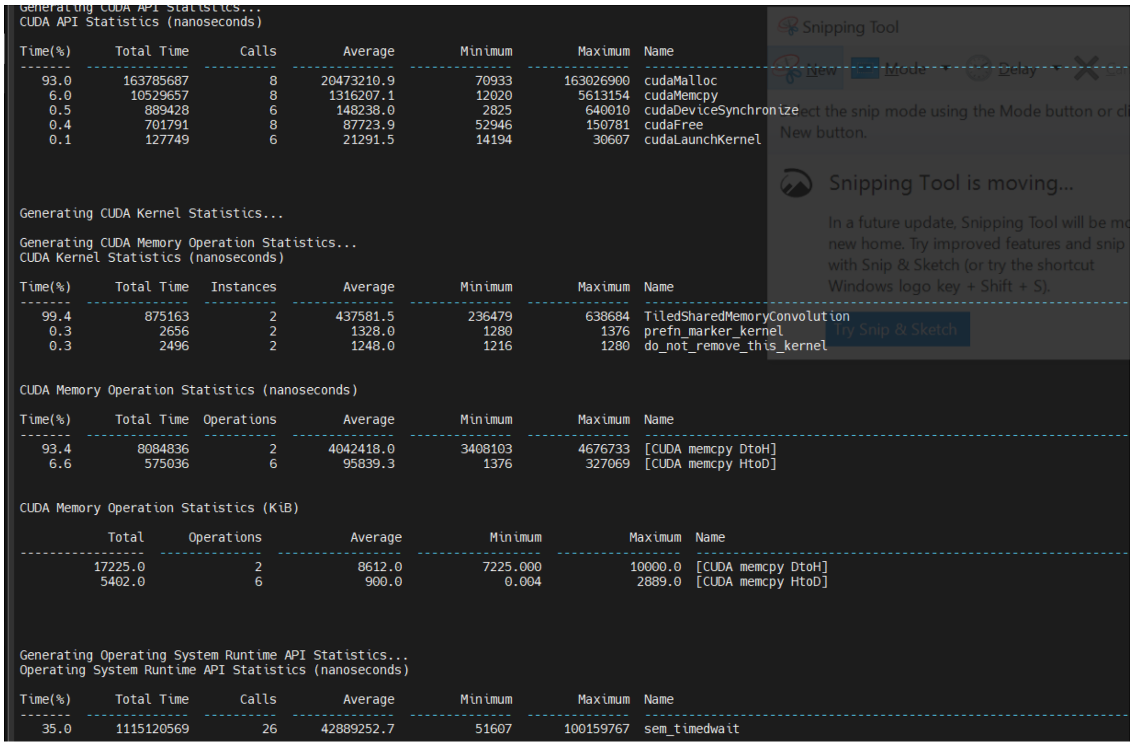
Batch Size	Op Time 1	Op Time 2	Total Execution	Accuracy

			Time	
100	*0.246051 ms*	*0.645248 ms*	*0m1.194s*	*0.86*
1000	*2.34191 ms*	*6.34975 ms*	*10.134s*	*0.886*
10000	*23.1343 ms*	*63.5048 ms*	*1m39.583s*	*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 was actually not successful in improving performance. After doing some research on the internet, I learned that if I am using the data once and there is no data reuse between various threads within a block, using shared memory may actually take more time than global memory. And the reason behind this is when you copy data from global memory to shared, it still classifies as a global transaction, which takes time. Reads are faster when accessing shared memory, but it doesn't really matter if you already had to read the memory once from global memory.

Nsys profiling results with batch size 100:


Snipping Tool
New
Mode
Delay
Select the snip mode using the Mode button or click the New button.
Snipping Tool is moving...
In a future update, Snipping Tool will be moving to a new home. Try improved features and snip with Snip & Sketch (or try the shortcut Windows logo key + Shift + S).
Try Snip & Sketch

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   93.0       163785687           8      20473210.9           70933       163026900  cudaMalloc
    6.0        10529657           8       1316207.1           12020         5613154  cudaMemcpy
    0.5          889428           6        148238.0            2825          640010  cudaDeviceSynchronize
    0.4          701791           8         87723.9           52946          150781  cudaFree
    0.1          127749           6         21291.5           14194           30607  cudaLaunchKernel




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   99.4          875163           2        437581.5          236479          638684  TiledSharedMemoryConvolution
    0.3            2656           2          1328.0            1280            1376  prefn_marker_kernel
    0.3            2496           2          1248.0            1216            1280  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   93.4         8084836           2       4042418.0         3408103         4676733  [CUDA memcpy DtoH]
    6.6          575036           6         95839.3            1376          327069  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------------------
          17225.0               2             8612.0           7225.000            10000.0  [CUDA memcpy DtoH]
           5402.0               6              900.0              0.004             2889.0  [CUDA memcpy HtoD]




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   35.0      1115120569          26      42889252.7           51607       100159767  sem_timedwait
```

As you can see from the Nsys profiling results (average time, total time, etc.) and the OP times/total execution times from the table above, the time performances either didn't really change much or became worse than the time performances prior to utilizing tiled shared memory convolution. For instance, the OP times for a batch size of 10000 for this implementation were 23.1343 ms and 63.5048 ms, but without this optimization, the OP times for the same batch size were 19.613 ms and 54.4653 ms.

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

   -Kirk and Hwu Chapter 16,

   -https://stackoverflow.com/questions/10250325/cuda-shared-memory-not-faster-than-global

6. **Optimization 5: Fixed point (FP16) arithmetic + sweeping various parameters to find the best values**
   (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 to do FP16 or fixed point arithmetic because floats are 32 bits long, and you don't necessarily need 32 bits to represent every single data value. I believed truncating down to 16 bits (in CUDA, this is called a "half" data type) would help cut down on extra time. Furthermore, I wanted to gain some more experience handling different data types (although this reason is not specifically related to increasing time performances).

   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?

Essentially, there are several float values and floating point operations throughout the main forward kernel. Instead of using floats, convert to __halfs, and then keep everything else the same. I believed truncating down to 16 bits/converting to __half data type would help cut down on extra time and increase the performance of my forward convolution because I wouldn't have to deal with potentially extraneous bits when using floats (which has 2x the number of bits as halfs). I did synergize this optimization with my first optimization: Sweeping various parameters to find the best values (i.e. TILE_WIDTH = 12).

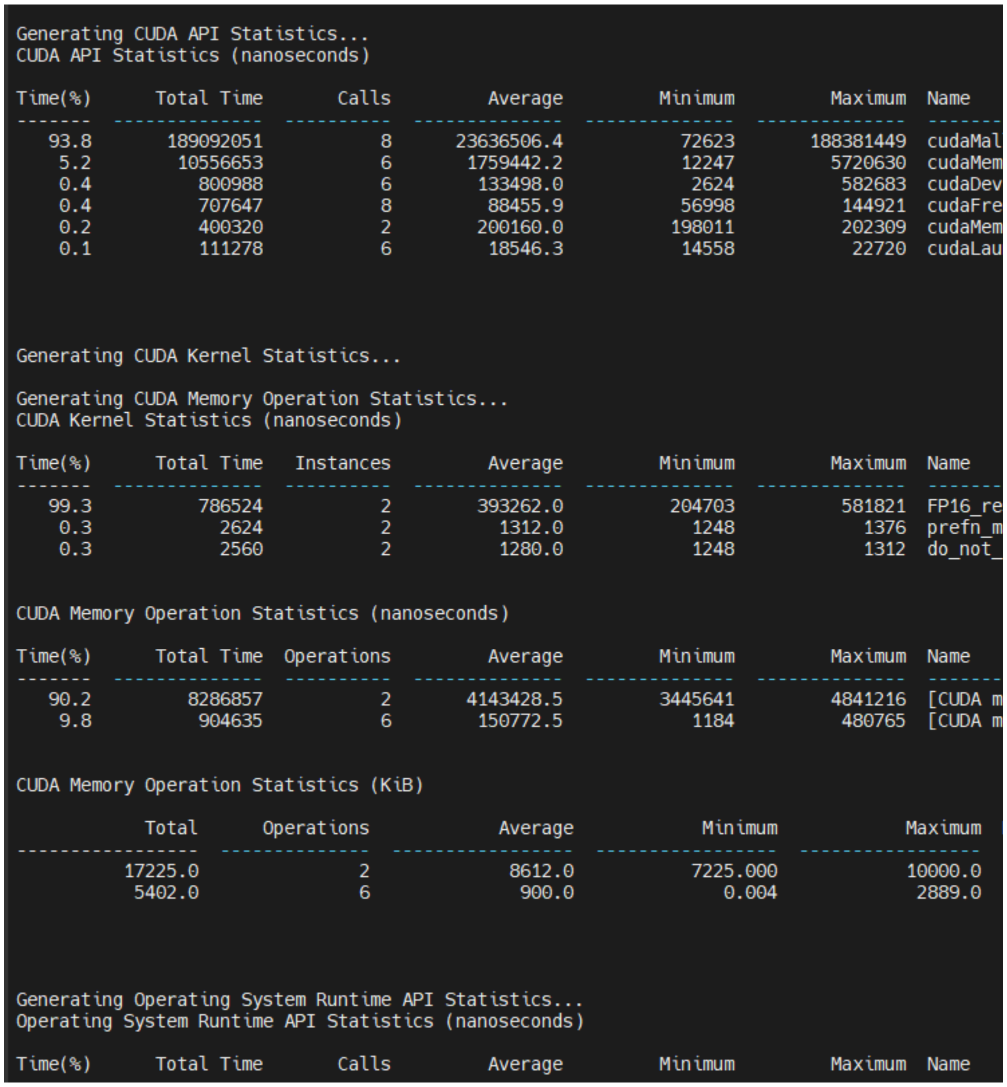
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.210636 ms*	*0.593005 ms*	*1.149s*	*0.86*
1000	*2.11955 ms*	*6.01826 ms*	*10.175s*	*0.886*
10000	*21.1560 ms*	*59.9124 ms*	*1m40.506s*	*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 did not appear to be very successful in improving performance. In fact, it appeared to have made the forward convolution slightly slower. The most likely cause behind this is that I had to use extra function calls, __float2half(), to convert some of the float values into halves, which can cost some extra overhead time. Furthermore, I can also deduce that converting from float to a different data type does not seem to change anything because the actual values, or methods of computation never changed, so the performance times should not change very drastically because of that factor.

Nsys profiling results with batch size 100:



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------
   93.8       189092051           8      23636506.4           72623       188381449  cudaMal
    5.2        10556653           6       1759442.2           12247         5720630  cudaMem
    0.4          800988           6        133498.0            2624          582683  cudaDev
    0.4          707647           8         88455.9           56998          144921  cudaFre
    0.2          400320           2        200160.0          198011          202309  cudaMem
    0.1          111278           6         18546.3           14558           22720  cudaLau




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------
   99.3          786524           2        393262.0          204703          581821  FP16_re
    0.3            2624           2          1312.0            1248            1376  prefn_m
    0.3            2560           2          1280.0            1248            1312  do_not_


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------
   90.2         8286857           2       4143428.5         3445641         4841216  [CUDA m
    9.8          904635           6        150772.5            1184          480765  [CUDA m


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum
-----------------  --------------  -----------------  -----------------  -----------------
          17225.0               2             8612.0           7225.000            10000.0
           5402.0               6              900.0              0.004             2889.0




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
```

As you can see from the Nsys profiling results (average time, total time, etc.) and the OP times/total execution times from the table above, the time performances either didn't really change much or became worse than the time performances prior to FP16 arithmetic. For instance, the OP times for a batch size of 10000 for this implementation were *21.1560 ms* and *59.9124 ms*, but without this optimization, the OP times for the same batch size were 19.613 ms and 54.4653 ms.

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

-*https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html* f.

g.

h.

i.