Project: CNN on GPU Design Competition

Contributors:

Nuoyan Wang

Disclaimer:

As this competition is a semesterly competition, to promote creativity and unique designs while preventing plagiarism, final codes should not be released online publicly, but can be provided in private if needed.

Description:

- **Utilized C and CUDA to implement and optimize a convolution neural net (CNN) for purposes of image processing in AI.**

- Applied GPU benefits on CNN with featuring variable strides, shared memory, streaming, and unrolling

- Placed in Top 20 on competition leaderboard, maxing available optimization points

Design Report

0: Baseline

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 for this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.185649 ms	0.890992 ms	real 0m3.901s user 0m1.421s sys 0m0.294s	0.86
1000	1.6977 ms	8.67731 ms	real 0m10.825s user 0m10.473s sys 0m0.352s	0.886
5000	8.45003 ms	43.548 ms	real 0m51.033s user 0m50.034s sys 0m1.009s	0.871

Optimization 1: FP16 Half Precision Conversion (4 points) *(1-final-fp_sixteen.cu)*

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

I started off by implementing the half precision conversion in which the input and mask values are converted to 16-bit __half values within the kernel code to hopefully save time on the numerous FLOPs being done in the kernel. I chose this optimization first because it seemed to be the most independent from other optimizations, this way I can gauge if my final fastest code should be done using 32-bit or 16-bit. Because it was ultimately unsuccessful, the remainder of my optimizations could all stick to 32-bit floats.

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?**

Here, the input and mask values are converted to 16-bit __half values before being multiplied together, and stored in the now edited __half accumulator variable. This would allow for faster computation speeds when doing the FLOPs involved in the CNN, because instead of repeatedly multiplying 32-bit floats from ***input*** and ***mask***, then adding this 32-bit result into the ***acc*** (accumulator), all of those FLOPs will be faster since they deal with FP16-bit __half values. Lastly, they are reverted to floats before output. As this is my first optimization, and seems to be less related to other factors such as tiling, unrolling, etc, it should be able to synergize if effective (unfortunately it wasn't).

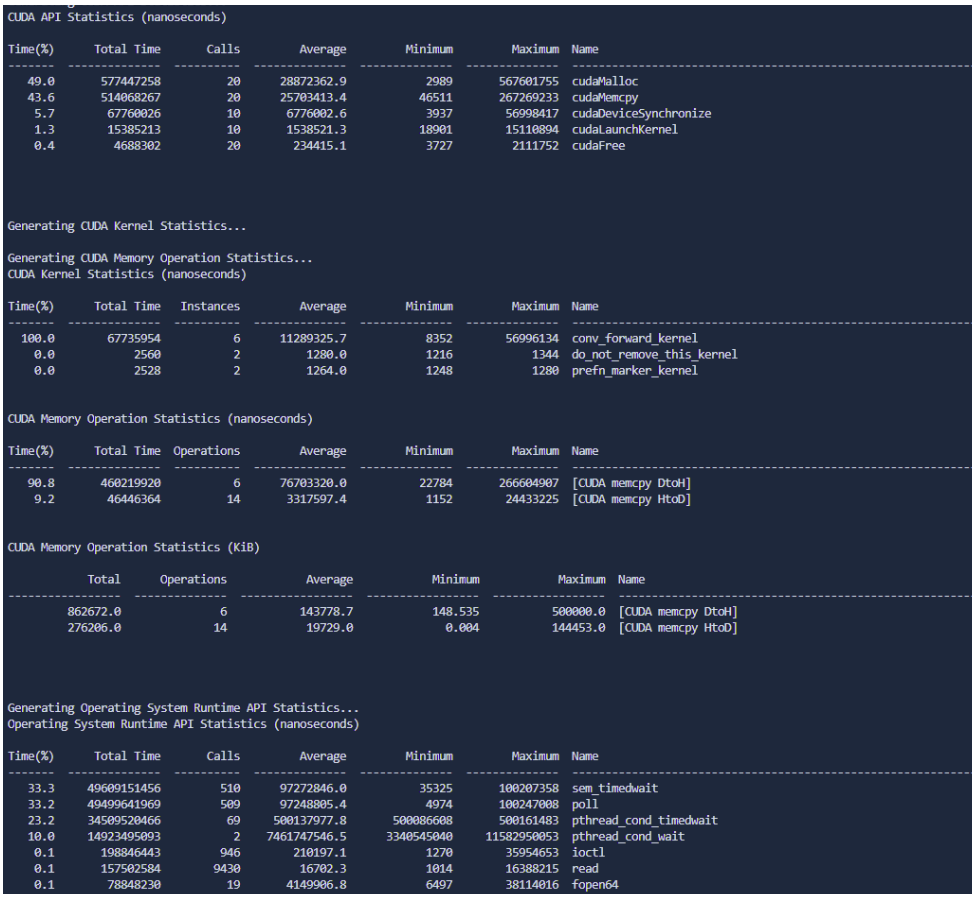
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used)

(Only FP16 optimization is used for values in table below)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.227661 ms	1.1588 ms	real 0m1.548s user 0m1.299s sys 0m0.220s	0.86
1000	2.15218 ms	11.415 ms	real 0m10.123s user 0m9.737s sys 0m0.348s	0.887
5000	10.6848 ms	56.9763 ms	real 0m51.033s user 0m50.034s sys 0m1.009s	0.8712

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 not successful in improving performance. My hypothesis that the 16-bit-half-point-OPs would be faster than 32-bit-FLOPs was likely offset by the amount of extra overhead required in the conversion of every input/mask value from float to half, and then re-converting the result back from half to float. Due to rounding that occurred in obtaining fp16, accuracy slightly deviates from what is expected.



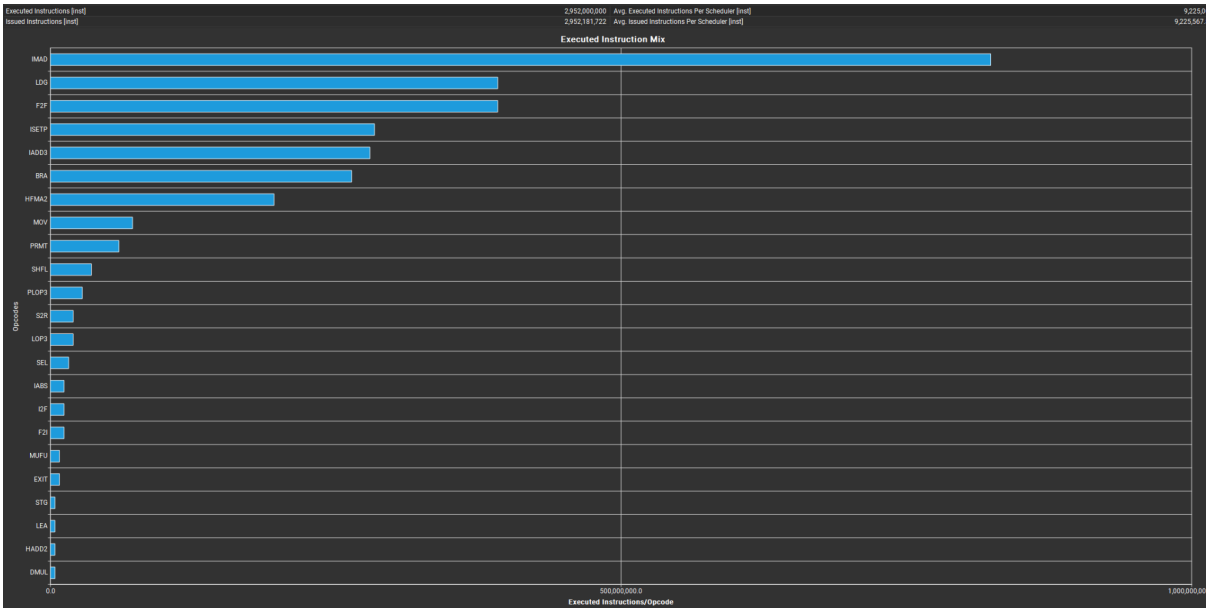
```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------------------
   49.0       577447258          20      28872362.9            2989       567601755  cudaMalloc
   43.6       514068267          20      25703413.4           46511       267269233  cudaMemcpy
    5.7        67760026          10       6776002.6            3937        56998417  cudaDeviceSynchronize
    1.3        15385213          10       1538521.3           18901        15110894  cudaLaunchKernel
    0.4         4688302          20        234415.1            3727         2111752  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------------------
  100.0        67735954           6      11289325.7            8352        56996134  conv_forward_kernel
    0.0            2560           2          1280.0            1216            1344  do_not_remove_this_kernel
    0.0            2528           2          1264.0            1248            1280  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------------------
   90.8       460219920           6      76703320.0           22784       266604907  [CUDA memcpy DtoH]
    9.2        46446364          14       3317597.4            1152        24433225  [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]



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------------------------------------
   33.3     49609151456         510      97272846.0           35325       100207358  sem_timedwait
   33.2     49499641969         509      97248805.4            4974       100247008  poll
   23.2     34509520466          69     500137977.8          500086608       500161483  pthread_cond_timedwait
   10.0     14923495093           2    7461747546.5         3340545040     11582950053  pthread_cond_wait
    0.1       198846443         946        210197.1            1270        35954653  ioctl
    0.1       157502584        9430         16702.3            1014        16388215  read
    0.1        78848230          19       4149906.8            6497        38114016  fopen64
```

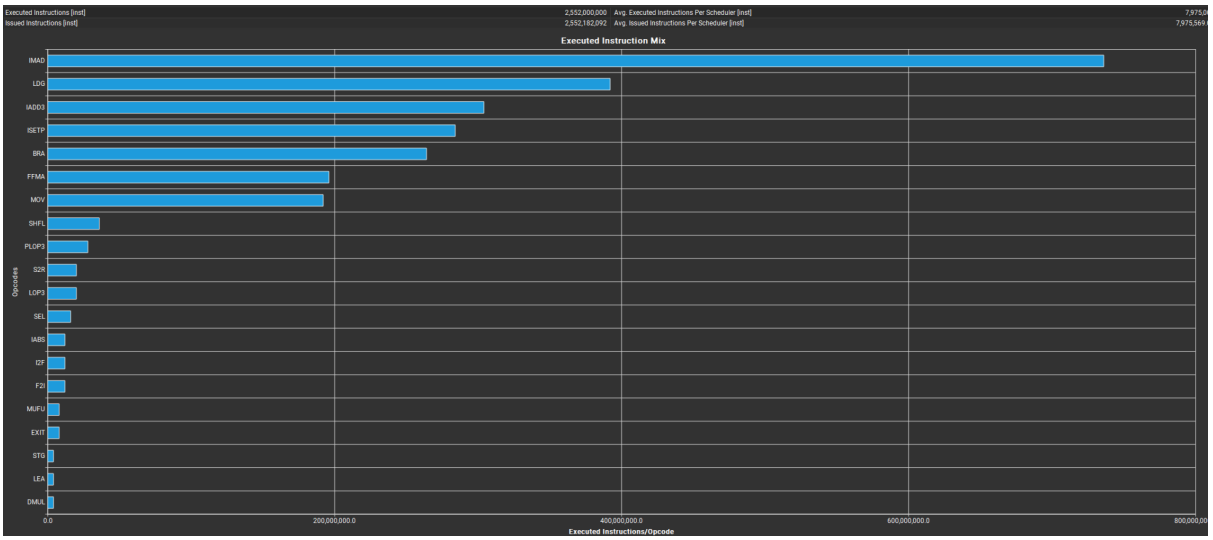
Notice the conv_forward_kernel now takes 67.735754ms compared to when running nsys on baseline code, which took only 52.097124ms (shown below is baseline's nsys)

CUDA Kernel Statistics (nanoseconds)

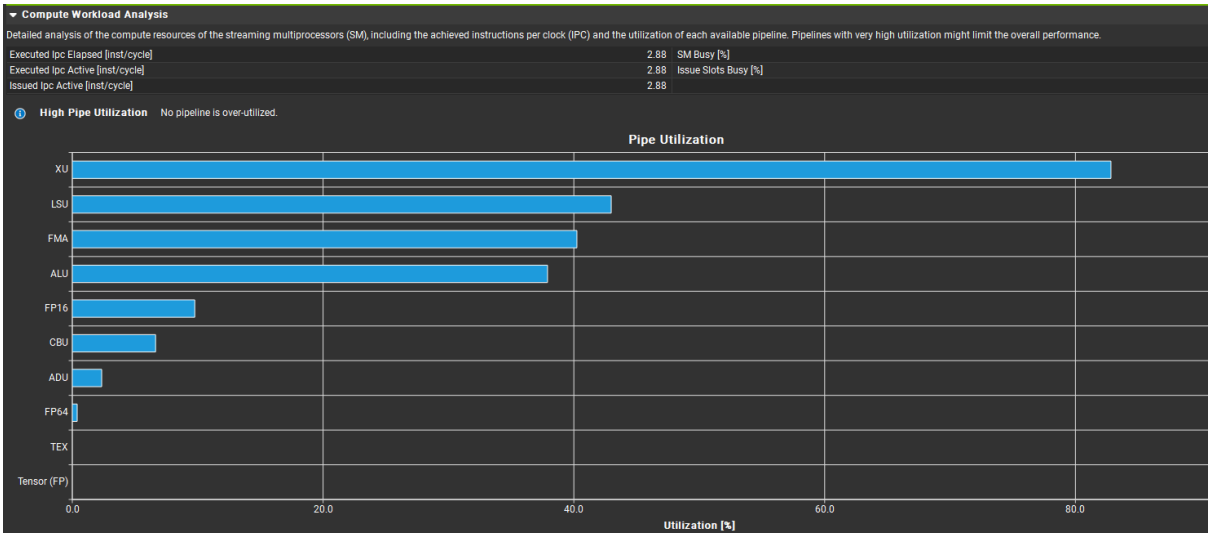
Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	52097124	6	8682854.0	7840	43611130	conv_forward_kernel

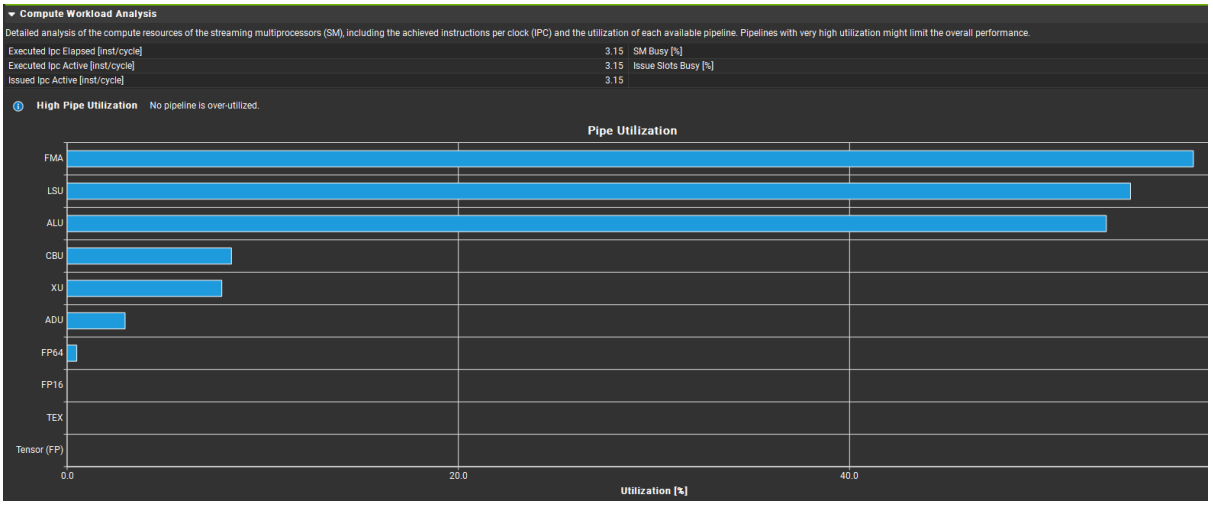
This is explained by the nsight analysis of the fp16 (top) vs the baseline (bottom). There are significantly more instructions performed by the fp16 kernel (400,000,000 or 15.6% more). As the graph indicates, everything else remains similar, but there is additional 'F2F' (float-to-float conversion) instructions in the fp16 implementation which simply did not occur in the baseline. These extra instructions offset the benefits given from the shortened arithmetic calculation time.





Similarly we see the fp16 creates tremendous more work for the XU which is responsible for the conversions. In fact, on the new optimization (top) compute workload analysis, the XU is doing the most work at 82% utilization, while in the baseline (bottom), the XU is only 7%. There is also a new amount of work done by FP16.





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

https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__HALF__MISC.html

https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

CampusWire ECE 408 #660, #636, #697

Optimization 2: Shared Memory Tiling (3 points) *(2-final-shared-mem.cu)*

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

I chose to implement shared memory convolution as that is something we had experience with in MP4. Moreover, it seemed like an optimization that would certainly improve performance because it is certain there will be significantly fewer global memory accesses. It will outweigh the overhead of loading into shared 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?

Instead of repeatedly accessing global memory for the input, we have an entire block use shared memory by first loading a chunk of dimension: C * tile_width * tile_width. Afterwards, threads within a block access this chunk of inputs from shared memory instead of global memory. Making much fewer global memory accesses should increase performance of forward convolution. As we have seen that optimization 1 (fp16) was not successful, this new optimization 2 is faster alone compared to when it is combined with op1 (fp16). As such, we will not synergize this shared memory tiling with other optimizations for now.

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)

(Only Convolution w/ Shared Memory optimization is used for values in table below)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.202182 ms	0.769426 ms	real 0m1.595s user 0m1.410s sys 0m0.152s	0.86
1000	1.83935 ms	7.42425 ms	real 0m10.997s user 0m10.541s sys 0m0.304s	0.886
5000	9.06529 ms	36.7719 ms	real 0m51.095s user 0m50.183s sys 0m0.856s	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).

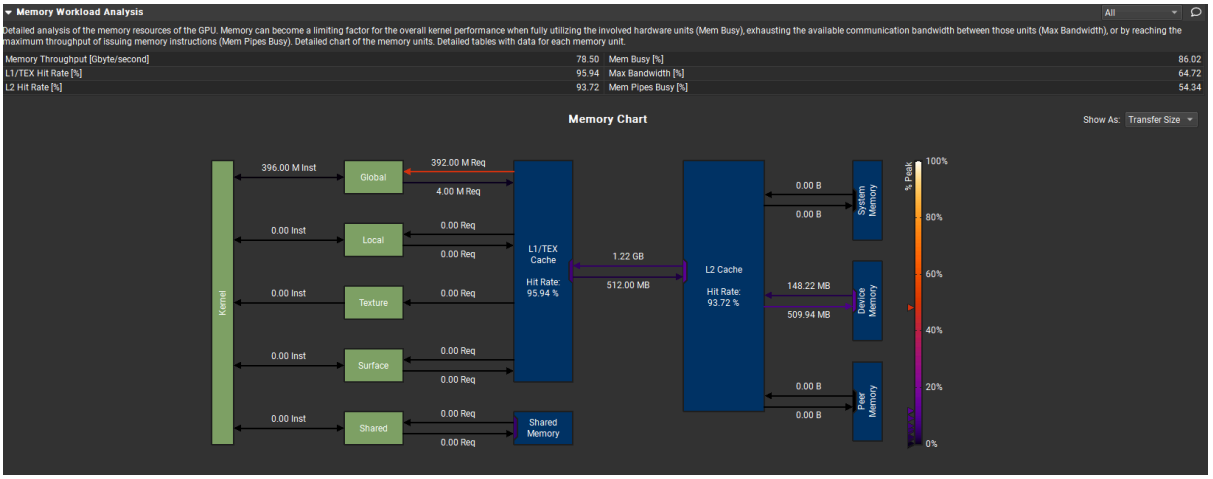
This was effective. We notice a shorter Total Time in conv_forward_kernel as compared to the baseline (46.354465ms vs 52.097124ms)

```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   67.5       528565344          20      26428267.2           29076       280442253  cudaMemcpy
   25.1       196323059          20       9816152.9            2355       193178572  cudaMalloc
    5.9        46380894          10       4638089.4            3331        37112481  cudaDeviceSynchronize
    1.2         9329788          10        932978.8           19945         9089913  cudaLaunchKernel
    0.3         2622614          20        131130.7            3972          459664  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        46354465           6       7725744.2           12547        37106504  conv_forward_kernel
    0.0            2817           2          1408.5            1376            1441  do_not_remove_this_kernel
    0.0            2529           2          1264.5            1216            1313  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   91.2       478638002           6      79773000.3           23141       279738289  [CUDA memcpy DtoH]
    8.8        45921856          14       3280132.6            1152        24012078  [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]



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   33.3     51845686945         533      97271457.7           41668       100394374  sem_timedwait
   33.3     51785228818         532      97340655.7           64379       100361514  poll
   22.2     34508516660          69     500123429.9       500048201       500173219  pthread_cond_timedwait
   11.1     17281634938           2    8640817469.0      4588327082     12693307856  pthread_cond_wait
    0.1        85651942         945         90637.0            1012        15585172  ioctl
```

Nsys for baseline (below)

```
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        52097124           6       8682854.0            7840        43611130  conv_forward_kernel
```

This can be explained by analyzing the reduction of global memory accesses in Nsight. We can see in the memory workload analysis that there is less data being transferred between kernel and global memory (216.50M vs 396.00M). Instead, we see new faster data transfers occurring between the kernel and shared memory (212.50M vs 0M).



Specifically looking at the shared me usage we see that we were originally not using any of it to our advantage, but this optimization greatly puts it to use.

Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	0	0	0	0	0
Shared Store	0	0			0
Shared Atomic	0	0	0	0	0

Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	196,000,000	196,000,000	398,215,243	45.58	198,664,001
Shared Store	16,500,000	16,500,000			214,435
Shared Atomic	0	0	0	0	0

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

Texting page 358 (shared memory tiling implementation)

Dynamic Shared Memory

https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

Optimization 3: Kernel in Constant Memory (0.5 points) *(3-final-const+shared.cu)*

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

I chose to implement shared memory convolution as that is something we again had experience with in MP4. Moreover, it seemed like an optimization that would certainly improve performance because again it is certain there will be significantly fewer global memory accesses.

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?

Instead of repeatedly accessing global memory for the kernel, we have a kernel stored in constant memory, as we know we will not need to be editing our change the kernel through the CNN process. I believe this would increase performance as there is a guaranteed significant reduction in global memory access as compared to repeatedly going to global memory for the mask. This optimization is able to synergize well with optimization 2 (shared memory), as we can see that the Op Times below when op2 and op3 are combined, are faster than either optimization 2 or 3 alone. Again, we do not attempt to synergize fp16.

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)

(Input in Shared Memory + Mask in Constant Memory BOTH used for values in table below)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.176979 ms	0.669708 ms	real 0m1.588s user 0m1.381s sys 0m0.200s	0.86
1000	1.61801 ms	6.45146 ms	real 0m10.586s user 0m10.340s sys 0m0.244s	0.886
5000	8.01837 ms	32.3147 ms	real 0m51.503s user 0m50.466s sys 0m0.868s	0.871

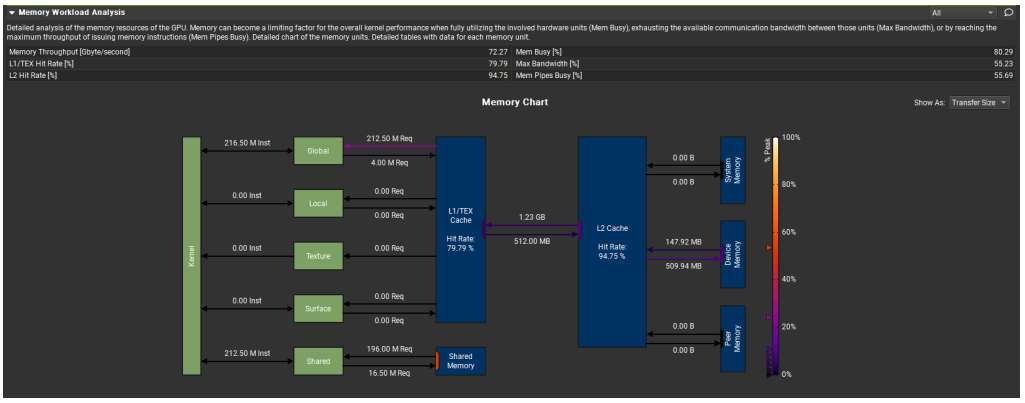
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

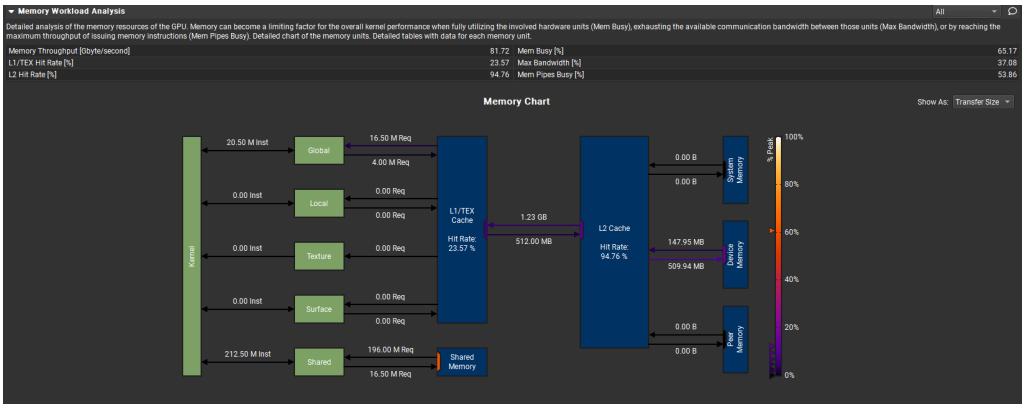
This was effective. See results from nsys below, where total time spent in conv_forward_kernel decreases from the results from optimization 1 of 46.354465ms to the new 40.432454ms.

```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   67.4       551576541          14      39398324.4           31055       297866543  cudaMemcpy
   22.0       180336025          20       9016801.2            2544       176093547  cudaMalloc
    4.9        40468179          10       4046817.9            3528        32250839  cudaDeviceSynchronize
    4.5        36551634          20       1827581.7            4020        33595074  cudaFree
    1.2         9526506          10        952650.6           25228         9266877  cudaLaunchKernel
    0.1          444251           6         74041.8           58005           88012  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        40432454           6       6738742.3           12000        32248364  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2496           2          1248.0            1216            1280  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   92.8       505878677           6      84313112.8           12705       297110624  [CUDA memcpy DtoH]
    7.2        39448074          14       2817719.6            1152        20718698  [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]



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   33.3     51783918690         533      97155569.8           20301       100351955  sem_timedwait
   33.3     51699603385         531      97362718.2           29616       100620714  poll
   21.9     34008589342          68     500126313.9       500063594       500501654  pthread_cond_timedwait
   11.4     17729300161           2    8864650080.5      4451226062     13278074099  pthread_cond_wait
    0.1       122179659         944        129427.6            1057        24504887  ioctl
    0.0        21249961        9426          2254.4            1064           18634  read
```

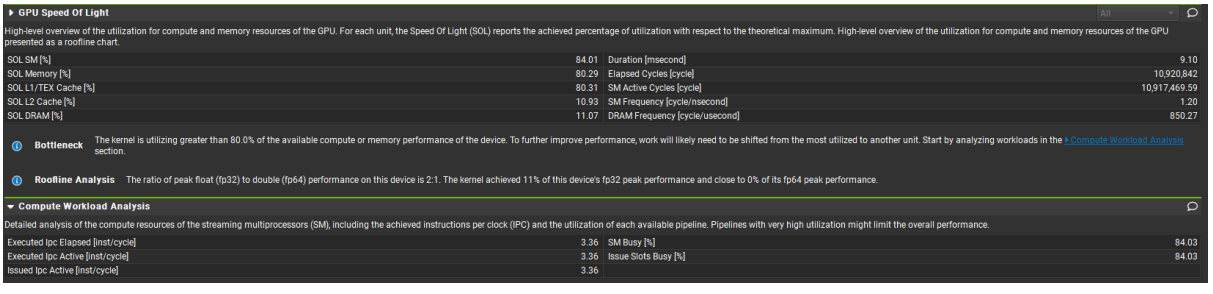
This can be explained by the memory workload analysis. Notice that our throughput increases from 72.27 GB/s to 81.72 GB/s, while the % of memory busy decreases from 80.29 % to 65.17 %. We can see from the graph that we are only accessing 20.50 M from global memory as compared to the previous 216.50 M. Combined, these indicate that memory access is now less of a bottleneck.

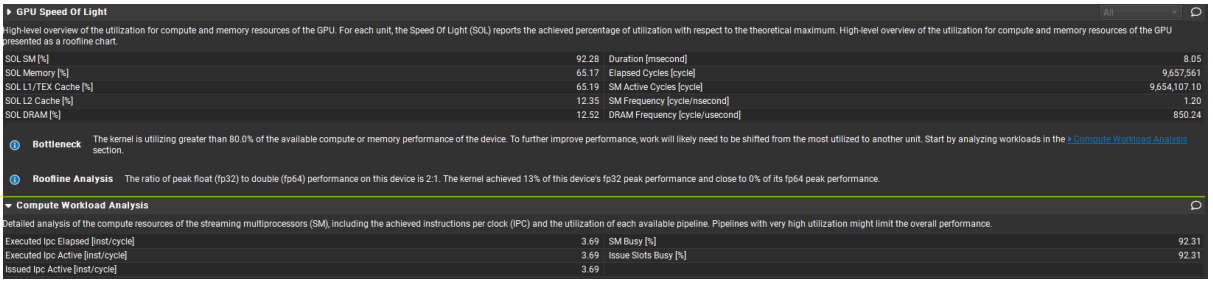
(Top image is from previous optimization with just shared memory, bottom adds constant memory)





As a result, we note that we have increased higher SOL SM % from 84.01 to 92.28, but lower SOL Memory % from 80.29 to 65.17. More SMs are busy (from 84.03% to 92.31%). This indicates more computation is able to be done, as opposed to being memory limited by repeated accesses to global memory in order to get the mask.





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

My MP4 code (constant memory masks)

Optimization 4: Streams for Overlapping Transfer & Computation (4 points)
(4-final-const+shared+streams.cu)

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

Next, I chose to implement streams in order to overlap data transfer and computation. I believe that since our implement is split into multiple batches, it would make sense that they can run separately in different streams, and this will lower total runtime.

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

The host function sees its input B and creates exactly B streams. For each stream, an adjusted portion of device memory is created, and CudaMemcpyAsynced to an adjusted pointer. Now the gridDim becomes (M, Y, B/num_streams) or (M, Y, 1). Each created kernel computes from that specific section B, and results are copied back to adjusted pointers. This should improve performance as we can now have kernels running code while other sections transfer data, just as explained in lecture 22.

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)

(Shared Memory + Constant Memory Mask + Stream Overlap used for values in table below)

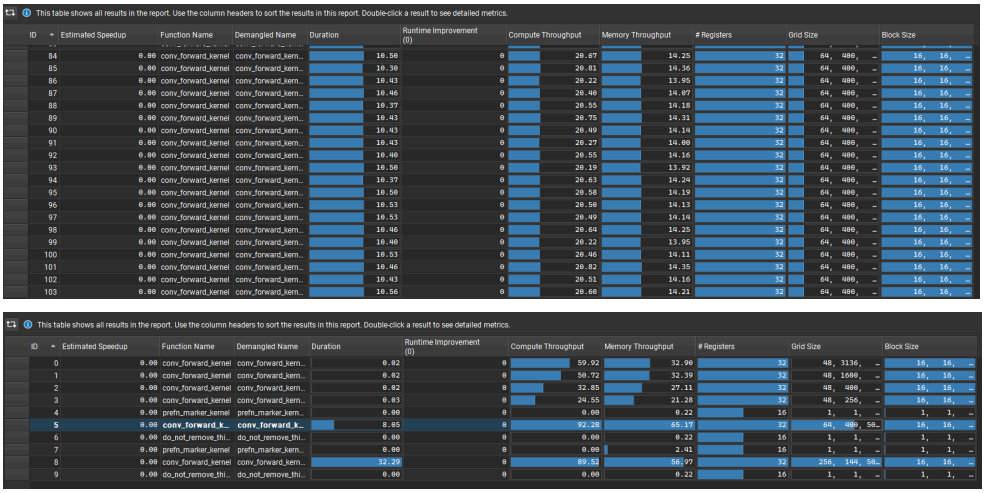
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.001278 ms	0.001977 ms	real 0m1.546s user 0m1.390s sys 0m0.152s	0.86
1000	0.007192 ms	0.01924 m	real 0m11.223s user 0m10.543s sys 0m0.360s	0.886
5000	0.14381 ms	0.630174 ms	real 1m0.656s user 0m58.069s sys 0m1.040s	0.871

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).

We successfully lowered our op-times, but the total execution did not get better. Notice in the nsys results below that the total time spent in conv_forward_kernel greatly increases from 40.432454ms to 266.941039ms.

```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   49.5      1508494290       10010        150698.7            1284       142153425  cudaStreamCreate
   28.4       864599145       20020         43186.8            5305          551591  cudaMemcpyAsync
   19.1       582428217          20      29121410.9            2277       191145027  cudaMalloc
    2.6        79941367       10014          7983.0            3877        25040502  cudaLaunchKernel
    0.2         4952841          20        247642.0            4152         1115742  cudaFree
    0.1         4127052          16        257940.8             818         1257998  cudaDeviceSynchronize
    0.1         3173584           2       1586792.0           47267         3126317  cudaMemcpy
    0.0          171666           6         28611.0           11070           44433  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
  100.0       266941039       10010         26667.4            6143           66080  conv_forward_kernel
    0.0            2560           2          1280.0            1248            1312  prefn_marker_kernel
    0.0            2528           2          1264.0            1248            1280  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   71.9       132573209       10010         13244.1            6048           89887  [CUDA memcpy DtoH]
   28.1        51839941       10018          5174.7            1408          179935  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

           Total      Operations           Average           Minimum           Maximum  Name
----------------  --------------  ----------------  ----------------  ----------------  ----------------------------------------
        862672.0           10010              86.2            42.188             577.0  [CUDA memcpy DtoH]
        276206.0           10018              27.6             0.004            1061.0  [CUDA memcpy HtoD]



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   33.0     54223464376         557      97349128.1           29449       100425178  sem_timedwait
   32.9     54200417845         556      97482765.9            6199       100429935  poll
   22.2     36509750286          73     500133565.6       500041208       500387528  pthread_cond_timedwait
   10.7     17634862769           2    8817431384.5      4271237781     13363624988  pthread_cond_wait
    1.2      1903880722        7043        270322.4            1000        76061215  ioctl
    0.0        23470037        9427          2489.7            1049          312247  read
```

It may seem confusing why the op times decreased so much, yet we spend much more time in the conv_forward_kernel. However, we know this is due to the staggered running of multiple kernels in streams, and this is reflected when we see the basic Nsight page. In the newest implementation, we have many more kernels, each one having a shorter staggered duration of (~10ms) while the older implementation contained two main working kernels of (8 + 32 ms). This leads to a shorter op-time within a given kernel, but significantly more total time spent in conv_forward_kernel.

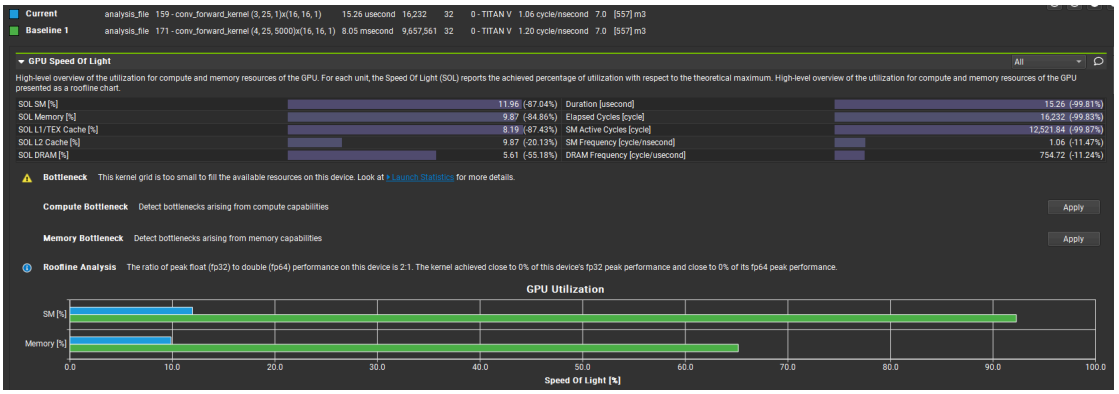


ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement (0)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
84	0.00	conv_forward_kernel	conv_forward_kern...	10.50	0	20.67	14.25	32	64, 400, ...	16, 16, ...
85	0.00	conv_forward_kernel	conv_forward_kern...	10.30	0	20.81	14.36	32	64, 400, ...	16, 16, ...
86	0.00	conv_forward_kernel	conv_forward_kern...	10.43	0	20.22	13.95	32	64, 400, ...	16, 16, ...
87	0.00	conv_forward_kernel	conv_forward_kern...	10.46	0	20.40	14.07	32	64, 400, ...	16, 16, ...
88	0.00	conv_forward_kernel	conv_forward_kern...	10.37	0	20.55	14.18	32	64, 400, ...	16, 16, ...
89	0.00	conv_forward_kernel	conv_forward_kern...	10.43	0	20.75	14.31	32	64, 400, ...	16, 16, ...
90	0.00	conv_forward_kernel	conv_forward_kern...	10.43	0	20.49	14.14	32	64, 400, ...	16, 16, ...
91	0.00	conv_forward_kernel	conv_forward_kern...	10.43	0	20.27	14.00	32	64, 400, ...	16, 16, ...
92	0.00	conv_forward_kernel	conv_forward_kern...	10.40	0	20.55	14.16	32	64, 400, ...	16, 16, ...
93	0.00	conv_forward_kernel	conv_forward_kern...	10.50	0	20.19	13.92	32	64, 400, ...	16, 16, ...
94	0.00	conv_forward_kernel	conv_forward_kern...	10.37	0	20.63	14.24	32	64, 400, ...	16, 16, ...
95	0.00	conv_forward_kernel	conv_forward_kern...	10.50	0	20.58	14.19	32	64, 400, ...	16, 16, ...
96	0.00	conv_forward_kernel	conv_forward_kern...	10.53	0	20.50	14.13	32	64, 400, ...	16, 16, ...
97	0.00	conv_forward_kernel	conv_forward_kern...	10.53	0	20.49	14.14	32	64, 400, ...	16, 16, ...
98	0.00	conv_forward_kernel	conv_forward_kern...	10.46	0	20.64	14.25	32	64, 400, ...	16, 16, ...
99	0.00	conv_forward_kernel	conv_forward_kern...	10.40	0	20.22	13.95	32	64, 400, ...	16, 16, ...
100	0.00	conv_forward_kernel	conv_forward_kern...	10.53	0	20.46	14.11	32	64, 400, ...	16, 16, ...
101	0.00	conv_forward_kernel	conv_forward_kern...	10.46	0	20.82	14.35	32	64, 400, ...	16, 16, ...
102	0.00	conv_forward_kernel	conv_forward_kern...	10.43	0	20.51	14.16	32	64, 400, ...	16, 16, ...
103	0.00	conv_forward_kernel	conv_forward_kern...	10.56	0	20.60	14.21	32	64, 400, ...	16, 16, ...

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement (0)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
0	0.00	conv_forward_kernel	conv_forward_kern...	0.02	0	59.92	32.90	32	48, 3136, ...	16, 16, ...
1	0.00	conv_forward_kernel	conv_forward_kern...	0.02	0	50.72	32.39	32	48, 1600, ...	16, 16, ...
2	0.00	conv_forward_kernel	conv_forward_kern...	0.02	0	32.85	27.11	32	48, 400, ...	16, 16, ...
3	0.00	conv_forward_kernel	conv_forward_kern...	0.03	0	24.55	21.28	32	48, 256, ...	16, 16, ...
4	0.00	prefn_marker_kernel	prefn_marker_kern...	0.00	0	0.00	0.22	16	1, 1, ...	1, 1, ...
5	0.00	conv_forward_k...	conv_forward_k...	8.85	0	92.28	65.17	32	64, 400, 50...	16, 16, ...
6	0.00	do_not_remove_thi...	do_not_remove_thi...	0.00	0	0.00	0.22	16	1, 1, ...	1, 1, ...
7	0.00	prefn_marker_kernel	prefn_marker_kern...	0.00	0	0.00	2.41	16	1, 1, ...	1, 1, ...
8	0.00	conv_forward_kernel	conv_forward_kern...	32.29	0	89.52	56.97	32	256, 144, 50...	16, 16, ...
9	0.00	do_not_remove_thi...	do_not_remove_thi...	0.00	0	0.00	0.22	16	1, 1, ...	1, 1, ...

We were unsuccessful in truly improving overall performance. This can be explained by numerous factors, such as the extra time spent creating multiple streams and kernels. If we look back at the nsys files, we see that we are now spending 1508.494290ms creating streams and ~79.9ms launching kernels, as compared to 0 ms and ~9.5ms respectively. Even with streams' advantage of overlap, this significant extra time contributes to a slower execution.

Lastly, if we directly compare the GPU utilization between our previous optimization (below called green baseline 1) with our newest stream optimization (below called blue current), we notice that each of our new kernels are doing significantly less work.



The new kernel takes 15.26 microseconds vs older one taking 8.05ms. However, it is obvious that the new streamed kernels are not being utilized to a full extent, as both their SM and Memory utilization percentages are much lower at ~10% each vs the non-streams 95% and 65%. We can conclude that the benefits of streams taking advantage of overlaps is offset by the sheer number of kernels we need to create which each do less work and are not using their full power. In other words, fewer kernels being maximized is still faster than numerous kernels doing little work.

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

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

Optimization 5: Tuning with __restrict__ + Loop Unroll (3 points) *(5-final-restrict-unroll.cu)*

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

Lastly, I chose to implement loop unrolling + tuning with restrict. I made this decision because I believed it was one of the optimizations that would not interfere with my previously made optimizations, as all of them still contained the same piece of code with the triple-for-loop across C*(mask).

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 accounts for the fact that for loops have inherent overhead and time involved, and thus we will manually code each line of the Kernel mask multiplication process, thus eliminating the overhead. Because this adjustment certainly provides the exact same functionality and behavior, it should be able to synergize very well.

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)

(Input in Shared Memory + Mask in Constant Memory + Unrolling are 3 optimizations all used for values in table below, but we did not include streaming)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.129326 ms	0.467645 ms	real 0m1.598s user 0m1.359s sys 0m0.181s	0.86
1000	1.13768 ms	4.41945 ms	real 0m10.545s user 0m10.162s sys 0m0.304s	0.886
5000	5.62252 ms	22.0636 ms	real 0m50.213s user 0m49.198s sys 0m0.952s	0.871

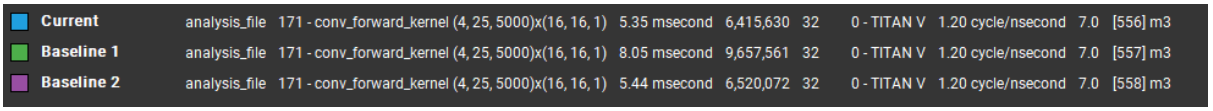
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Since we did not include streams, we are simply adding unrolling on top of our results from optimization 3. We now have shared memory, constant memory mask, and loop unrolling. We see that our time spent in conv_forward_kernel again decreases from 40.432454ms to 28.143481ms.

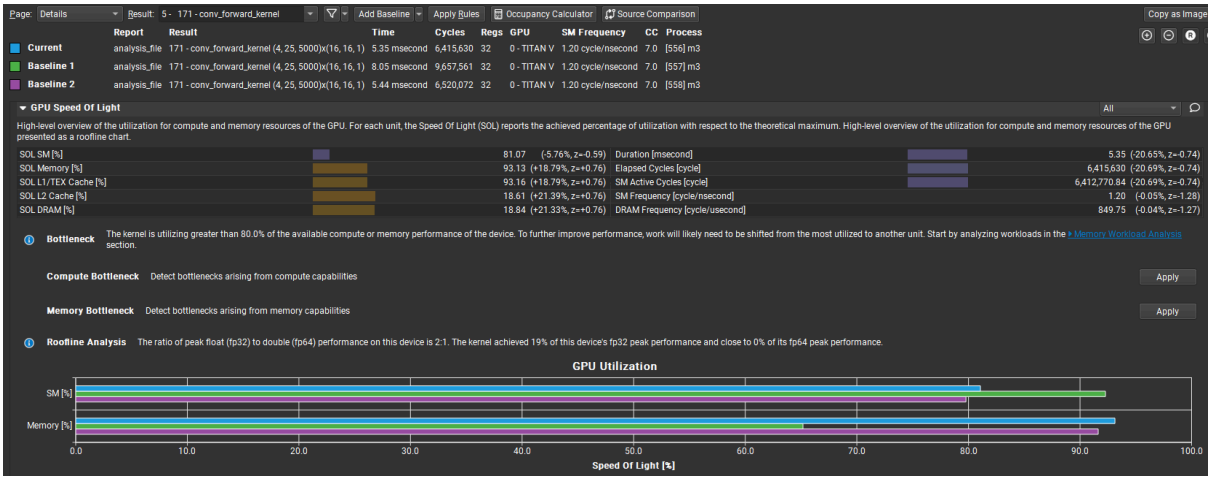
```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   70.2       541934956          14      38709639.7           29694       291922920  cudaMemcpy
   23.6       182353044          20       9117652.2            2203       178828070  cudaMalloc
    3.6        28167755          10       2816775.5            3524        22802236  cudaDeviceSynchronize
    2.1        16402596          10       1640259.6           23372        16161928  cudaLaunchKernel
    0.4         2706513          20        135325.6            4360          436568  cudaFree
    0.1          468104           6         78017.3           59626           88404  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        28143481           6       4690580.2           10400        22797983  conv_forward_kernel
    0.0            2848           2          1424.0            1408            1440  do_not_remove_this_kernel
    0.0            2688           2          1344.0            1312            1376  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   92.9       493293634           6      82215605.7           12672       291156856  [CUDA memcpy DtoH]
    7.1        37583480          14       2684534.3            1120        19576470  [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]



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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   33.3     50810008420         523      97151067.7           44063       100400590  sem_timedwait
   33.3     50793626578         522      97305798.0           57266       100297064  poll
   22.3     34009829933          68     500144557.8       500095230       500229759  pthread_cond_timedwait
   11.0     16770282064           2    8385141032.0      4088616125     12681665939  pthread_cond_wait
    0.1       104531833         942        110968.0            1145        17732071  ioctl
    0.0        21131219        9427          2241.6            1020           18678  read
```

Our results in Nsight can again explain the improvement in op-times and execution times. If we see below results, all 3 results have shared memory + constant memory mask, but additionally:

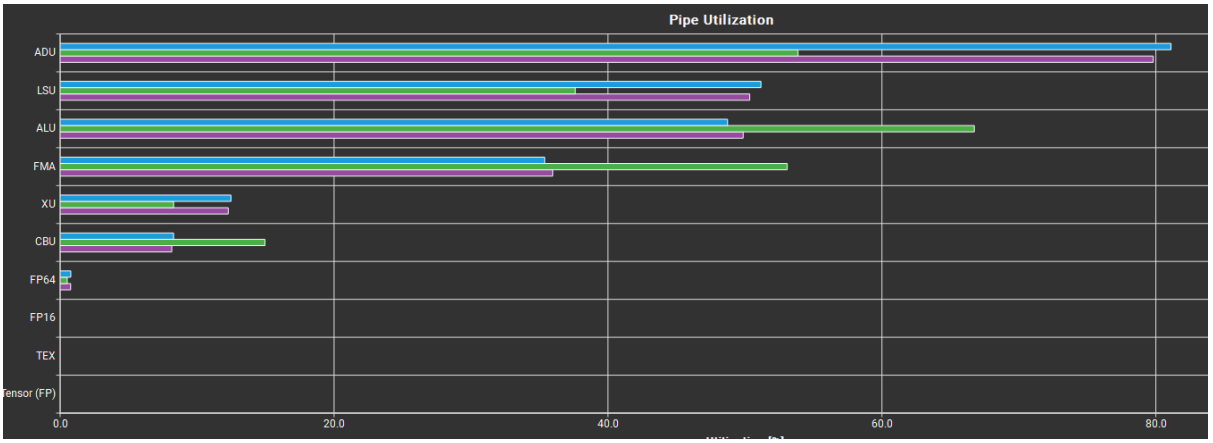
(green = no-unrolling, purple = unrolling when K=2,4,6, blue = unrolling when K = 1,3,5,7)



a) Note that without unrolling, we are certainly running many more cycles and spending significantly more time (8.05 ms vs ~5.5 ms).



b) We note that unrolling allows us to use a lower percentage of SM with a higher percentage of memory. This perhaps indicates the decrease in calculations for for-loop overhead, allowing for this to be less memory-bound.



c) Lastly, we notice the jump in pipe utilizations. In particular, note the lower usage of ALU and FMA in calculating for-loop overhead for both arithmetic calculations and comparisons.

Now, to discuss why I specifically chose to unroll when K = 1, 3, 5, 7, this is due to our testing. When only unrolling K = 1, 2, 3, 4, we do not achieve significant improvements. Again, if we choose to unroll for large K = 4, 5, 6, 7, we do not achieve a noticeable difference. Op-times sum to ~42 ms.

Next, I decided to exploit the nature of even/odd mask sizes, as I know there is a different behavior when convolving with an even length mask. When unrolling for even valued K, we again get an optime ~41ms. Lastly, when unrolling odd K = 1, 3, 5, 7, we manage to improve our optime to ~28ms.

This is likely due to the fact that odd length masks do not require the rounding or padding when convolving, and thus the unrolling of for loops make a more significant impact on the overage times.

These results are proven in the above Nsight results, where the unrolling of K = 1,3,5,7 (blue) allows for higher GPU utilization in both the SM % and Memory %.

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

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

   https://www.geeksforgeeks.org/loop-unrolling/