Name: *Li Zihao*
NetID: *zihaoli5*
Section: *ZJ1/ZJ2*

ECE 408/CS483 Milestone 3 Report

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.317849ms*	*1.19352ms*	*1.212s*	*0.86*
1000	*3.07987ms*	*12.1131ms*	*10.077s*	*0.886*
10000	*65.8809ms*	*109.786ms*	*1m40.669s*	*0.8714*

1. **Optimization 1: *coalesced memory access***

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

I choose to implement the memory-coalesced version of convolution.
In my baseline, I used threadIdx.x for the h-axis (vertical axis) and threadIdx.y for the w-axis (horizontal axis). In this type, the threads in a warp access different rows of the input x, which is not coalesced.
Then I decided my first optimization to turn the memory access into a coalesced type. I made some change that now in a block, threadIdx.x increases horizontally, and threadIdx.y increases vertically. In this pattern, the memory access for a warp of threads can be done within only several bursts.

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

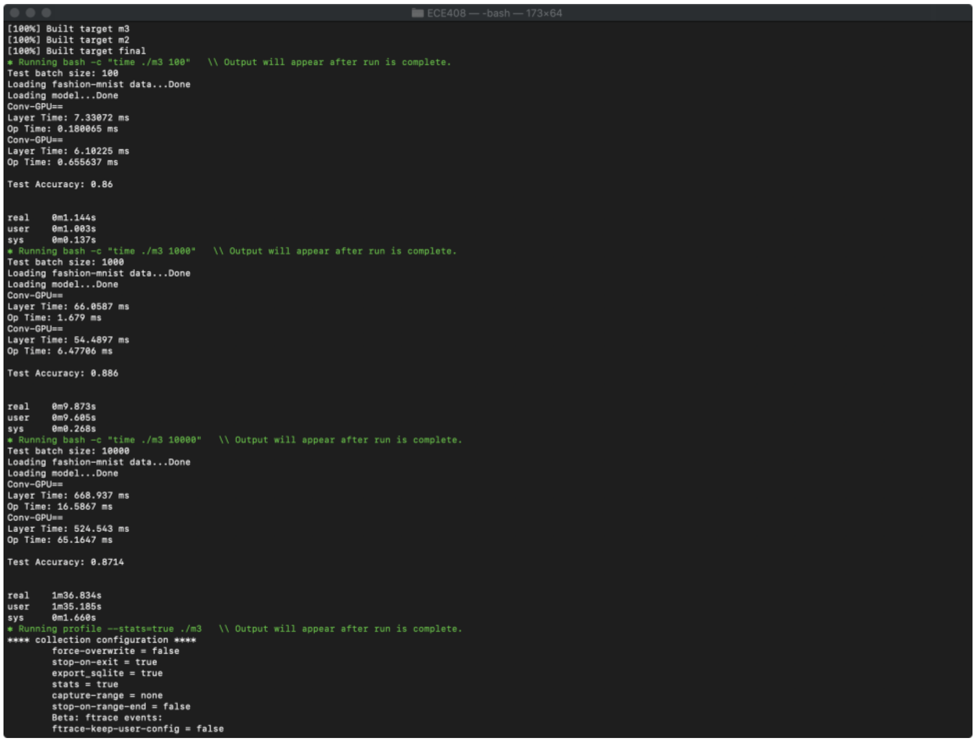
This optimization works because it can reduce the time of memory bursts. I think this optimization would increase the performance of the forward convolution because there is nothing different except that the direction of memory access is changed. This optimization does not synergize with any other previous optimizations, since this is the first optimization that I'm trying to implement.

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	*0.179219*	*0.655022 ms*	*0m1.150s*	*0.86*
1000	*1.67881 ms*	*6.48344 ms*	*0m9.575s*	*0.886*
10000	*16.59ms*	*65.157ms*	*1m39.241s*	*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 successfully improved the performance, as shown in the figure below:

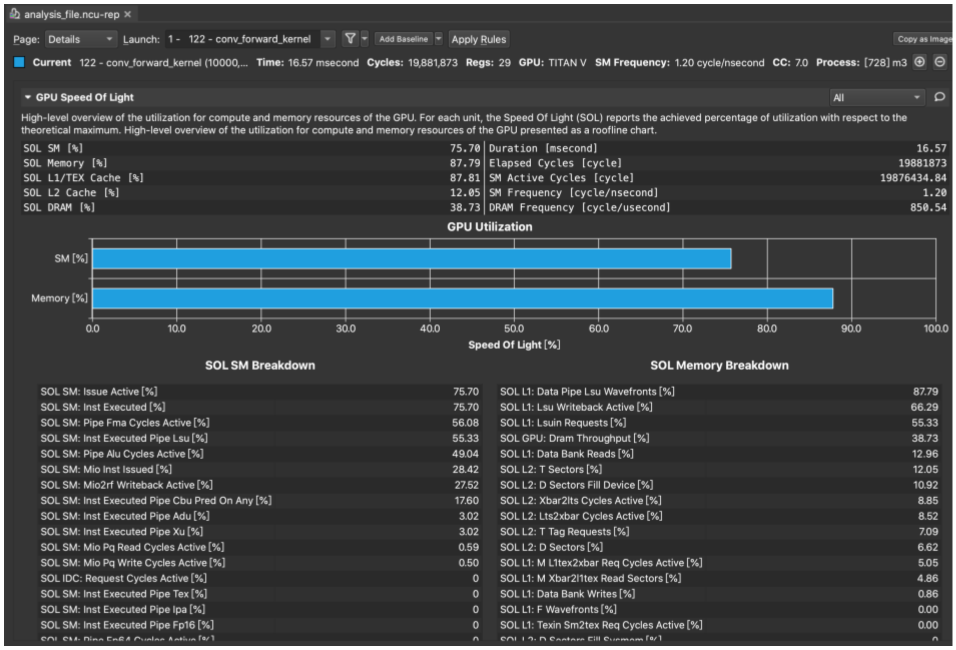


```
ECE408 — -bash — 173×64
[100%] Built target m3
[100%] Built target m2
[100%] Built target final
* Running bash -c "time ./m3 100"   \\ Output will appear after run is complete.
Test batch size: 100
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 7.33072 ms
Op Time: 0.180065 ms
Conv-GPU==
Layer Time: 6.10225 ms
Op Time: 0.655637 ms

Test Accuracy: 0.86


real    0m1.144s
user    0m1.003s
sys     0m0.137s
* Running bash -c "time ./m3 1000"   \\ Output will appear after run is complete.
Test batch size: 1000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 66.0587 ms
Op Time: 1.679 ms
Conv-GPU==
Layer Time: 54.4897 ms
Op Time: 6.47706 ms

Test Accuracy: 0.886


real    0m9.873s
user    0m9.605s
sys     0m0.268s
* Running bash -c "time ./m3 10000"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 668.937 ms
Op Time: 16.5867 ms
Conv-GPU==
Layer Time: 524.543 ms
Op Time: 65.1647 ms

Test Accuracy: 0.8714


real    1m36.834s
user    1m35.185s
sys     0m1.660s
* Running profile --stats=true ./m3   \\ Output will appear after run is complete.
**** collection configuration ****
        force-overwrite = false
        stop-on-exit = true
        export_sqlite = true
        stats = true
        capture-range = none
        stop-on-range-end = false
        Beta: ftrace events:
        ftrace-keep-user-config = false
```

the nsys result is shown as follows:

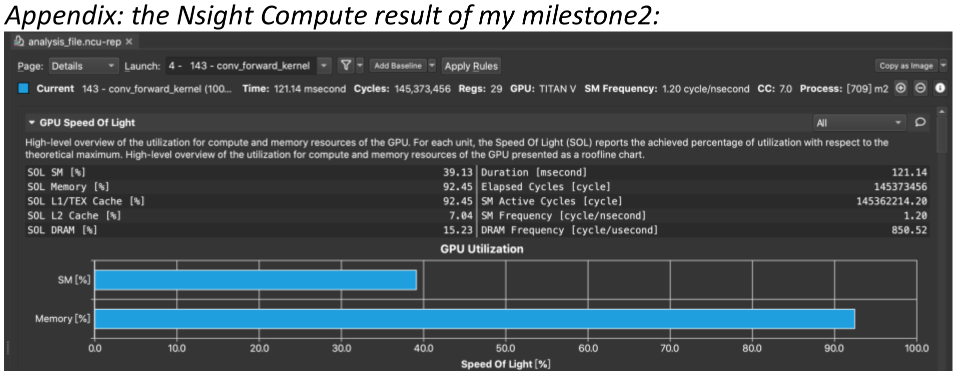
```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649685 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   80.1      1111559691           8     138944961.4           21884       608646177  cudaMemcpy
   12.6       174940072           8      21867509.0           76647       171535428  cudaMalloc
    5.9        82033946           6      13672324.3            3032        65309774  cudaDeviceSynchronize
    1.2        16279943           6       2713323.8           16322        16155344  cudaLaunchKernel
    0.2         2604121           8        325515.1           65493          924114  cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        82011531           2      41005765.5        16704371        65307160  conv_forward_kernel
    0.0            2688           2          1344.0            1344            1344  prefn_marker_kernel
    0.0            2656           2          1328.0            1312            1344  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   91.9      1017090732           2     508545366.0       409296762       607793970  [CUDA memcpy DtoH]
    8.1        89544716           6      14924119.3            1504        47992491  [CUDA memcpy HtoD]
```

We can see that the total time of conv_forward_kernel is 82.01ms, which is a lot faster than that of my milestone 2, which is 147.09ms.
As for the reason, we can see from the profiling result of Nsight Compute:



By using coalesced memory access, the efficiency of SM increased significantly, resulting in a faster convolution time.

Appendix: the Nsight Compute result of my milestone2:



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

Course lecture & textbook.

2. Optimization 2: *Weight matrix (kernel values) in constant memory (1 point)*

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

I choose to implement the optimization: Weight matrix (kernel values) in constant memory. I choose this optimization for my second optimization because this optimization is relatively easy to synergize with other optimizations. Also, this is an easy optimization. What I need to do is to declare constant memory for the kernel values and then use cudaMemcpyToSymbol.

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 work because the access to the constant memory is much faster than the global memory. I think this optimization would increase performance of the forward convolution because we only need to store the kernel value to the constant memory once and then the constant kernel value will be accessed a lot of times. This optimization synergizes with the previous optimization: ***coalesced memory access***

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

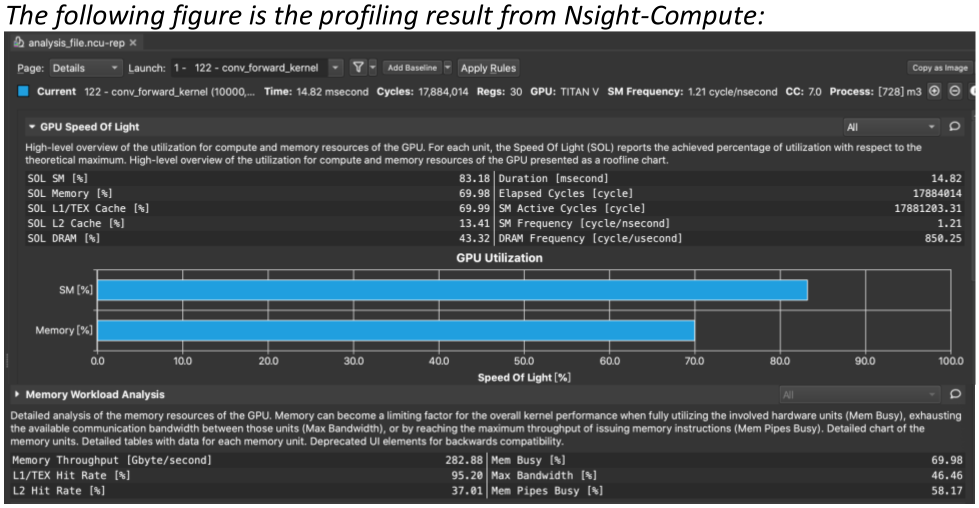
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.51148ms*	*5.72097ms*	*0m1.153s*	*0.86*
1000	*14.9241ms*	*57.8413ms*	*0m9.827s*	*0.886*
10000	*14.9241ms*	*57.8413ms*	*1m39.175s*	*0.8714*

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 does improve the performance successfully. This is because the access to kernel value speeds up much.
We can see from the nsys profiling result below that the time for conv_forward_kernel decreases to 73.02ms.

```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649590 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   78.8      1119327033           6     186554505.5           18709       612592776  cudaMemcpy
   14.2       201496368           6      33582728.0          321429       196698653  cudaMalloc
    5.1        73054268           6      12175711.3            2843        58013135  cudaDeviceSynchronize
    1.6        22923517           6       3820586.2           16495        22797728  cudaLaunchKernel
    0.2         3405632           6        567605.3           89421         1412769  cudaFree
    0.0          167038           2         83519.0           82134           84904  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        73028115           2      36514057.5        15016679        58011436  conv_forward_kernel
    0.0            2944           2          1472.0            1440            1504  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
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   92.9      1023613845           2     511806922.5       411976194       611637651  [CUDA memcpy DtoH]
    7.1        77823468           6      12970578.0            1536        41055348  [CUDA memcpy HtoD]
```

The following figure is the profiling result from Nsight-Compute:



It turns out that the SM is even more efficient, compared to the result for optimization 1. The efficiency of memory access is also improved, compared to that of a single optimization of coalescing:

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit. Deprecated UI elements for backwards compatibility.

Memory Throughput [Gbyte/second]	252.97	Mem Busy [%]	87.79
L1/TEX Hit Rate [%]	95.90	Max Bandwidth [%]	66.29
L2 Hit Rate [%]	37.07	Mem Pipes Busy [%]	55.33

The memory throughput of coalesced+constant_kernel is 282.88, which is larger than that of only coalesced, 252.97. This results in the improvement on performance.

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

Course lecture & textbook.

3. Optimization 3: *Tiled shared memory convolution (2 points)*

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

I choose to implement the optimization: Tiled shared memory convolution (2 points). I think I am familiar with this optimization, since this is an optimization that was taught in the very beginning of this class.

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?

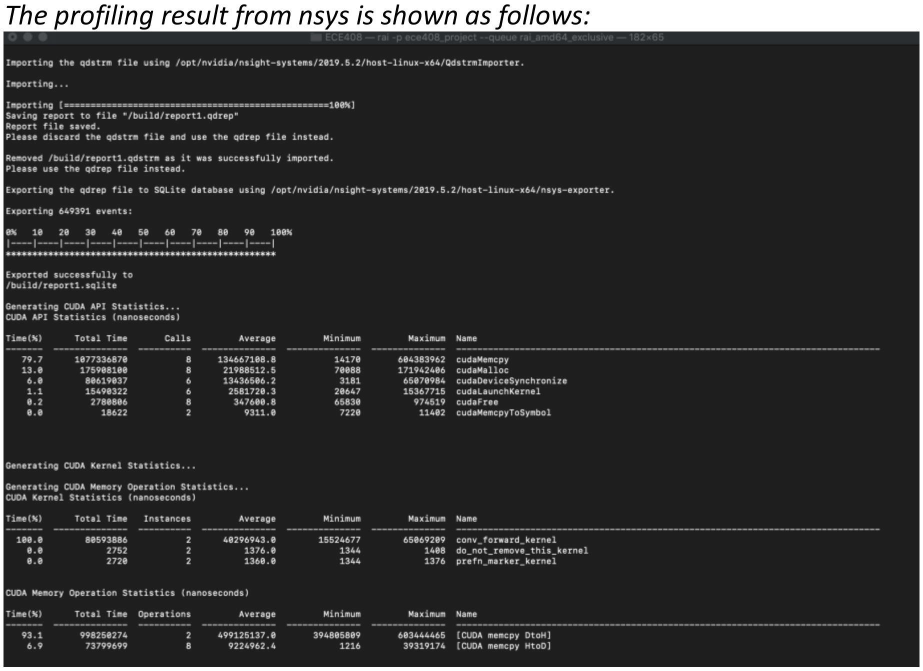
As discussed in the lecture, if we can use shared memory appropriate, theoretically we can increase memory reuse. Initially I think this optimization would increase the performance of the forward convolution, since accessing to the shared memory is usually faster than global memory. However, when testing I found that this optimization actually cannot improve the performance. This optimization synergizes with the previous 2 optimizations: coalesced memory access and 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 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.164951ms*	*0.654448ms*	*0m1.158s*	*0.86*
1000	*1.57406ms*	*6.5213ms*	*0m10.030s*	*0.886*
10000	*15.4801ms*	*64.9901ms*	*1m38.502s*	*0.8714*

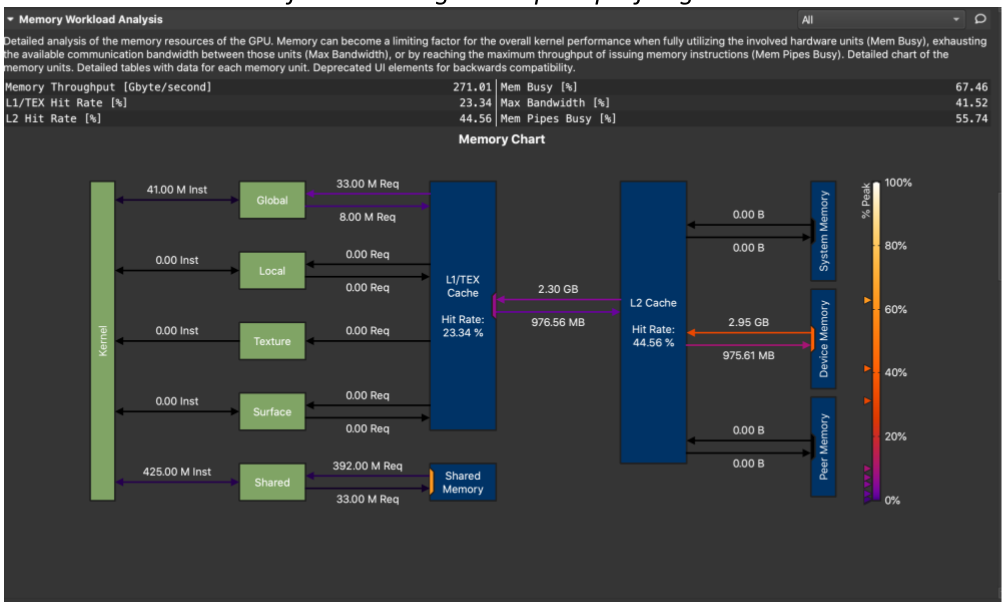
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization fails to improve performance.
The profiling result from nsys is shown as follows:



```
ECE408 — rai -p ece408_project --queue rai_amd64_exclusive — 182×65

Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649391 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   79.7      1077336870           8     134667108.8           14170       604383962  cudaMemcpy
   13.0       175908100           8      21988512.5           70088       171942406  cudaMalloc
    6.0        80619037           6      13436506.2            3181        65070984  cudaDeviceSynchronize
    1.1        15490322           6       2581720.3           20647        15367715  cudaLaunchKernel
    0.2         2780806           8        347600.8           65830          974519  cudaFree
    0.0           18622           2          9311.0            7220           11402  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        80593886           2      40296943.0        15524677        65069209  conv_forward_kernel
    0.0            2752           2          1376.0            1344            1408  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
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   93.1       998250274           2     499125137.0       394805809       603444465  [CUDA memcpy DtoH]
    6.9        73799699           8       9224962.4            1216        39319174  [CUDA memcpy HtoD]
```

As shown, the time for conv_forward_kernel increases to 80.59ms, which is larger than the previous 73.02ms.
The reason can be seen from the Nsight Compute profiling result:



What we can see is that, it is true that the access to shared memory is faster than global memory. However, the difference is not that large, and the declaration of shared memory consumes time:

Shared Memory	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	392000000	392000000	798009729	53.46	398721247
Shared Store	33000000	33000000	33000000	0.55	687506
Shared Atomic	0	0	0	0	0
Other	-	-	108687556	8.94	332
Total	425000000	425000000	939697285	62.95	399409085

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

Course lecture & textbook.
Usage of dynamic shared memory: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

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

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

I choose to implement the optimization: Input channel reduction: atomics. I choose this because I think this optimization technique can synergizes with my previous optimizations: coalesced memory access, Weight matrix (kernel values) in constant memory and Tiled shared memory convolution.

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?

   Before implementing atomic reduction on the input channel, my conv_forward_kernel uses a thread for each element on each output feature map of each batch. This requires one thread to compute the convolution of multiple input feature maps with their corresponding kernel values. Using atomic reduction, for each output element on an output feature map, for each input-output pair, a thread will be used to calculate the convolution values. Then we will let C (number of input channel) threads put their value to the appropriate place one by one.
   Originally, I thought this optimization would increase performance, since the computation of different input channel can be done at the same time. However, I found that this optimization actually cannot increase performance when testing.
   This optimization synergizes with 3 of my previous optimizations: coalesced memory access, Weight matrix (kernel values) in constant memory and Tiled shared memory convolution.

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

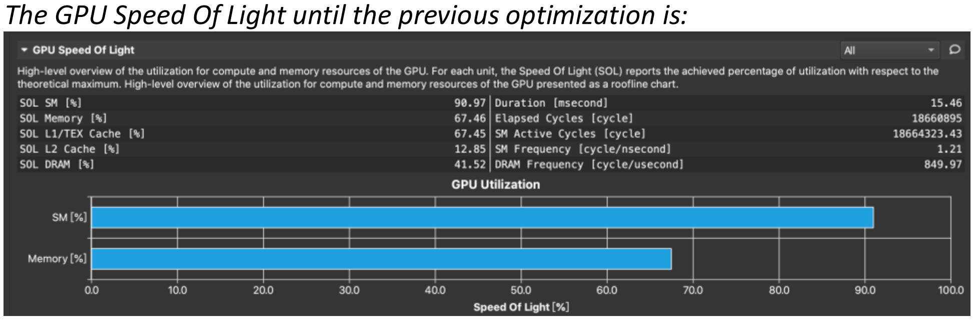
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.16612ms*	*0.760874ms*	*0m1.143s*	*0.86*
1000	*1.61044ms*	*7.98854ms*	*0m9.953s*	*0.886*
10000	*15.9807ms*	*79.5769ms*	*1m38.070s*	*0.8714*

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).

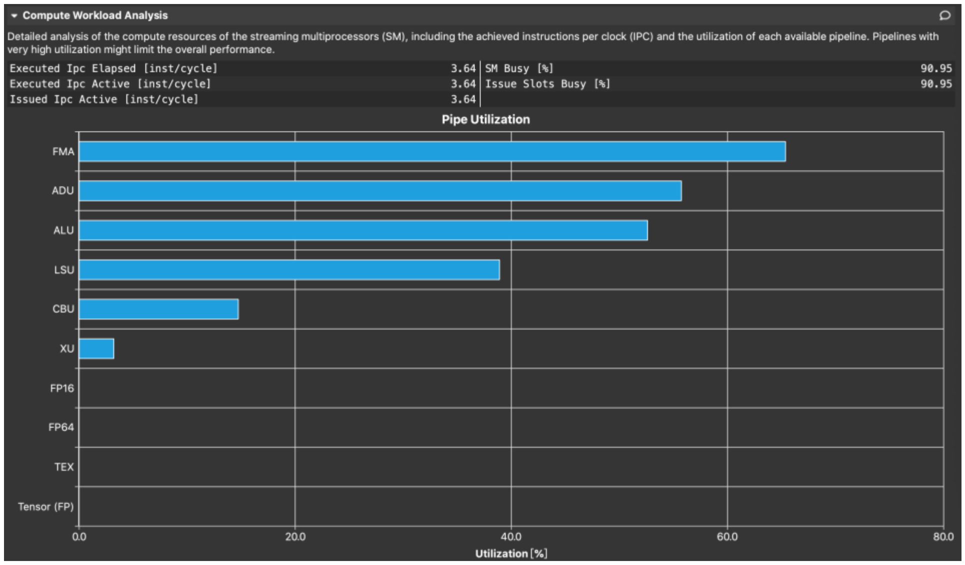
This optimization fails to improve performance.
The profiling result from nsys is shown as follows:

```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649479 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   77.8      1017139868           8     127142483.5           16494       536297668  cudaMemcpy
   12.9       168688652           8      21086081.5           69077       163949419  cudaMalloc
    7.4        96433589           6      16072264.8            2983        80373617  cudaDeviceSynchronize
    1.7        21951996           6       3658666.0           14694        21837584  cudaLaunchKernel
    0.2         2693691           8        336711.4           63024          839266  cudaFree
    0.0           18788           2          9394.0            7308           11480  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0        96399096           2      48199548.0        16035796        80363300  conv_forward_kernel
    0.0            2816           2          1408.0            1280            1536  prefn_marker_kernel
    0.0            2720           2          1360.0            1344            1376  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   92.9       931763855           2     465881927.5       396283812       535480043  [CUDA memcpy DtoH]
    7.1        71696380           8       8962047.5            1216        36159069  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538932.0               8            67366.0              0.004           288906.0  [CUDA memcpy HtoD]
```

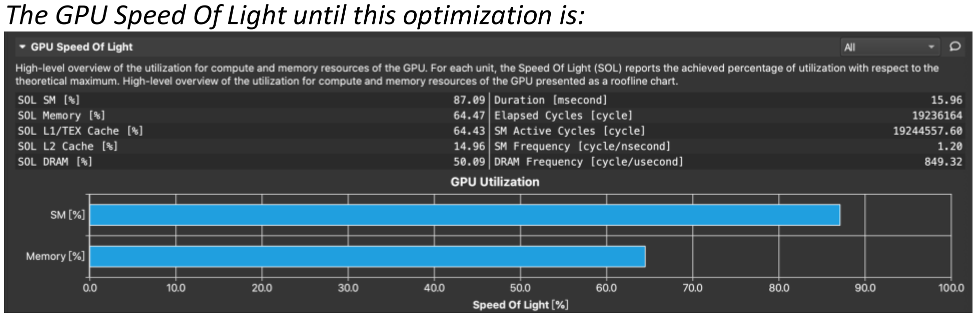
As shown, the time for conv_forward_kernel is 93.176ms, which is larger than that of the previous optimization. Before adding atomic reduction, the timing result is 80.59ms.
The reason can be seen from the Nsight Compute profiling result:
The GPU Speed Of Light until the previous optimization is:



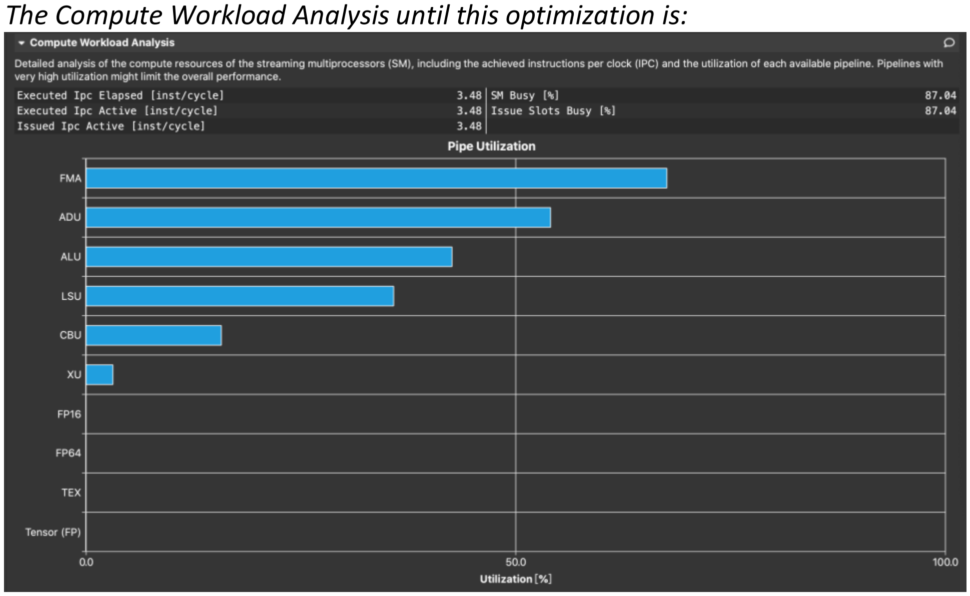
The Compute Workload Analysis until previous optimization is:



The GPU Speed Of Light until this optimization is:



The Compute Workload Analysis until this optimization is:



We can see that after implementation of atomicAdd(), the Speed Of Light of SM and Memory both decreases. Also, the compute workload decreases a little.
Actually, though atomic reduction can make it possible to compute different input channel in parallel, it has to use the function atomicAdd(). The call for that function and waiting atomically consume time, resulting in an even longer op time.

Another possible reason why the atomic version is slower is that the memory access for a block with size (TILE_WIDTH, TILE_WIDTH, C) is not that coalesced. A warp will require several bursts on different input channel when executing.

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

   Course lecture & textbook.

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

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

I choose to implement the optimization: Input channel reduction: tree. In my previous optimization I implemented atomic reduction, so I choose to implement a similar optimization: tree reduction.

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

This optimization works similarly to the atomic reduction.
Before implementing tree reduction on the input channel, my conv_forward_kernel uses a thread for each element on each output feature map of each batch. This requires one thread to compute the convolution of multiple input feature maps with their corresponding kernel values. Using tree reduction, for each output element on an output feature map, for each input-output pair, a thread will be used to calculate the convolution values. Then we add the results from C (number of input channel) threads together using a pattern of reduction tree and put the final result into the appropriate place.
Originally, I thought this optimization would increase performance, since the computation of different input channel can be done at the same time. However, I found that this optimization actually cannot increase performance when testing.
This optimization synergizes with the previous 2 optimizations: coalesced memory access and 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 10k images using this optimization (including any previous optimizations also used).

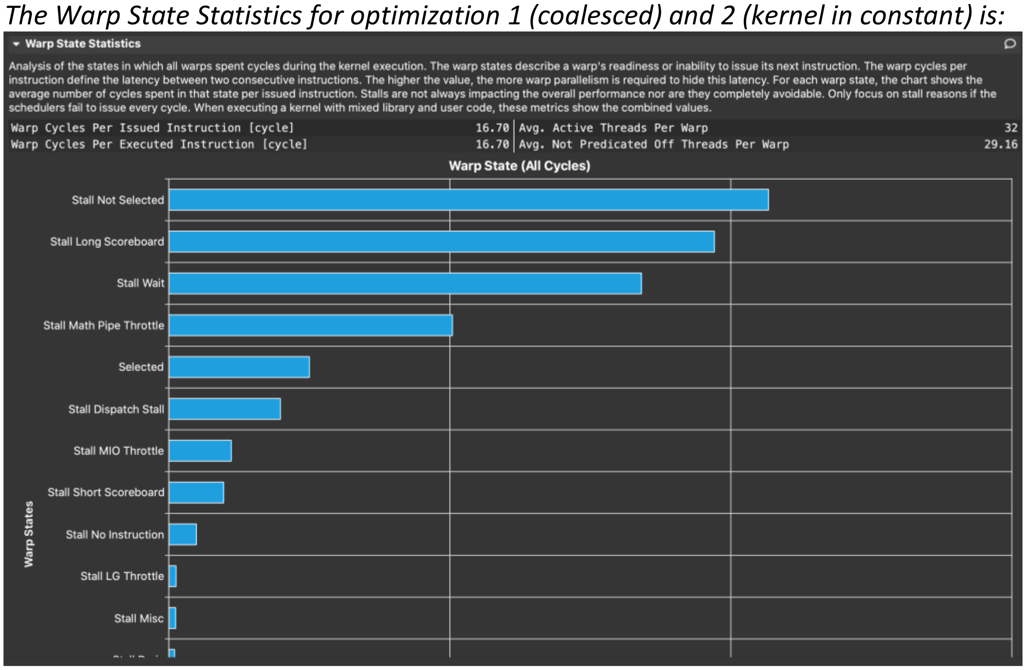
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.164584ms*	*0.818015*	*0m4.410s*	*0.86*
1000	*1.59745ms*	*8.72305ms*	*0m12.330s*	*0.886*
10000	*15.8308ms*	*87.4953ms*	*1m38.967s*	*0.8714*

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).

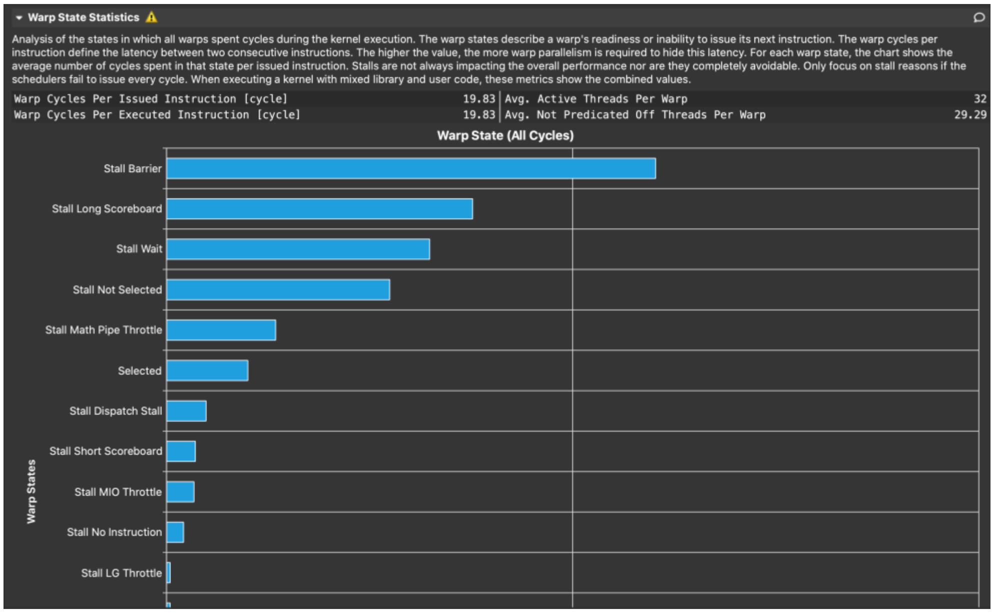
This optimization fails to improve performance.
The profiling result from nsys is shown as follows:

```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649574 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   77.8      1021872969           6     170312161.5           11479       538189922  cudaMemcpy
   12.8       168521846           6      28086974.3          283289       165275912  cudaMalloc
    7.9       104356174           6      17392695.7            3114        88416079  cudaDeviceSynchronize
    1.2        15573076           6       2595512.7           16888        15421548  cudaLaunchKernel
    0.2         2627429           6        437904.8           83251          935428  cudaFree
    0.0          350776           2        175388.0          174823          175953  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------
  100.0       104332997           2      52166498.5        15919185        88413812  conv_forward_kernel
    0.0            2912           2          1456.0            1408            1504  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
-------  --------------  ----------  --------------  --------------  --------------  --------------------
   91.2       927813455           2     463906727.5       390403308       537410147  [CUDA memcpy DtoH]
    8.8        89518573           6      14919762.2            1472        47991792  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]
```

As shown, the time for conv_forward_kernel is 93.176ms, which is larger than that of the previous optimization. Before adding tree reduction, the timing result is 73.02ms.
The reason can be seen from the Nsight Compute profiling result:
The Warp State Statistics for optimization 1 (coalesced) and 2 (kernel in constant) is:



However, after implementing tree reduction the Warp State Statistics become:



Since C is a relatively small number, tree reduction will cause warp divergence. Also, the declaration of shared memory consumes time. These are possible reasons that the tree reduction fails to improve performance.

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

   Course lecture & textbook.

6. **Optimization 6: *Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (1 point)***

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

      I choose the optimization: Sweeping various parameter to find best values for block sizes. This is an easy optimization and can synergize with my other optimization.

   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?

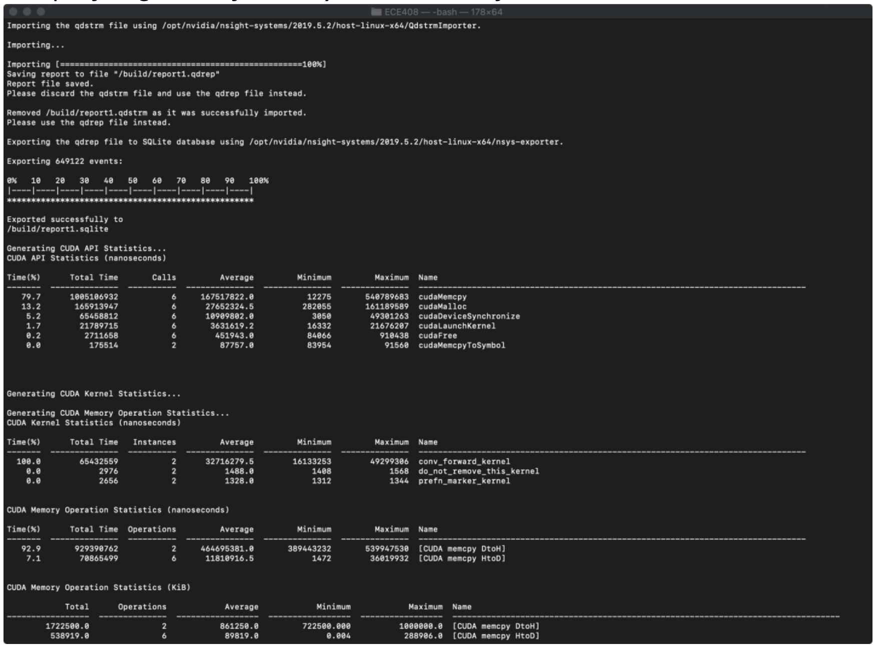
In my milestone2, I used TILE_WIDTH 16. However, the best value may not be 16, since the width and height of the output feature map may not be a multiple of 16. By sweeping various possible values of TILE_WIDTH, we can just pick the TILE_WIDTH that minimized the time for further use.
I think this optimization would increase performance of the forward convolution because there are various of numbers for me to sweep. Probably there is a number better than 16.
This optimization synergizes with 2 of my previous optimizations: coalesced memory access and 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 10k images using this optimization (including any previous optimizations also used).

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

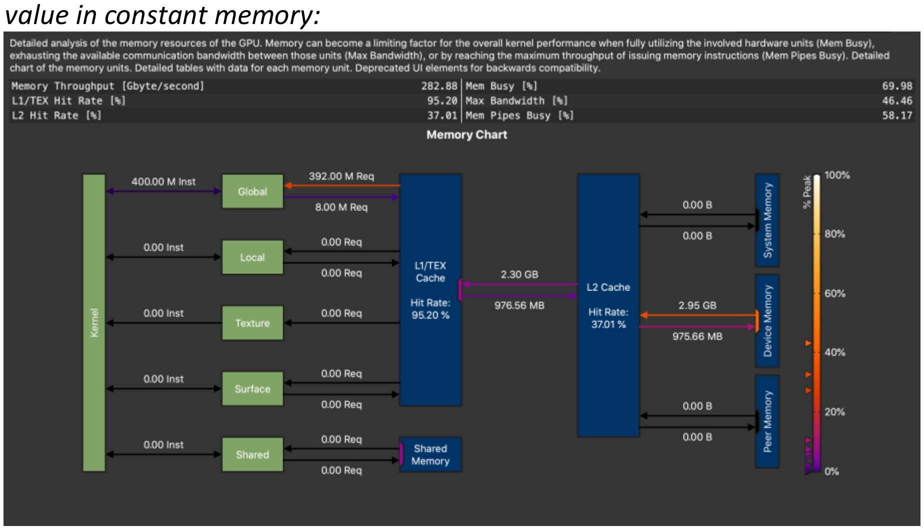
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization successfully improves the performance.
The profiling result from nsys is shown as follows:

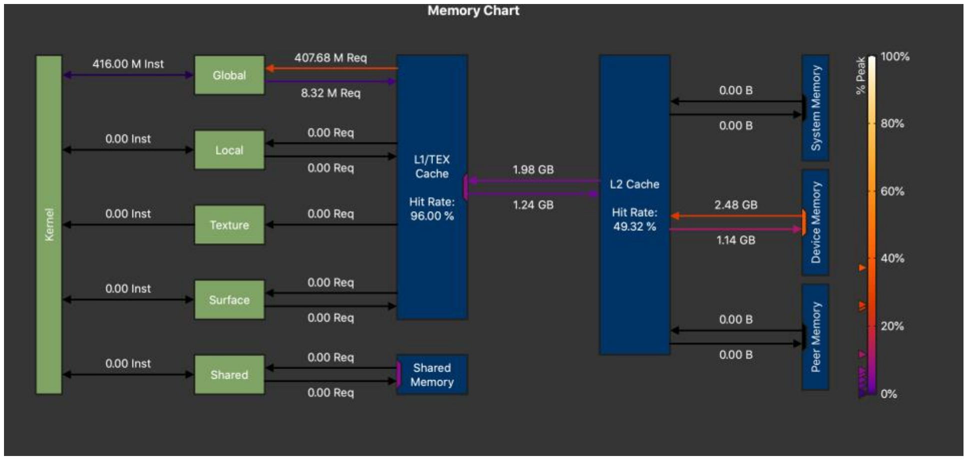


```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649122 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   79.7      1005106932           6     167517822.0           12275       540789683  cudaMemcpy
   13.2       165913947           6      27652324.5          282055       161189589  cudaMalloc
    5.2        65458812           6      10909802.0            3050        49301263  cudaDeviceSynchronize
    1.7        21789715           6       3631619.2           16332        21676207  cudaLaunchKernel
    0.2         2711658           6        451943.0           84066          910438  cudaFree
    0.0          175514           2         87757.0           83954           91560  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
  100.0        65432559           2      32716279.5        16133253        49299306  conv_forward_kernel
    0.0            2976           2          1488.0            1408            1568  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1312            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   92.9       929390762           2     464695381.0       389443232       539947530  [CUDA memcpy DtoH]
    7.1        70865499           6      11810916.5            1472        36019932  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------------------------------------------------------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]
```

As shown, the time for conv_forward_kernel is 65.43ms, which is smaller. Before adding this optimization, the timing result is 73.02ms.
The reason can be seen from the Nsight Compute profiling result:
This is the Memory Workload analysis for only coalesced memory access and kernel value in constant memory:



This is the Memory Workload analysis after changing TILE_WIDTH from 16 to 20:



We can see from the two charts that the total memory transition needed is smaller and Hit Rate of L2 cache increases from 37.01% to 49.32%. This means TILE_WIDTH 20 is more efficient than TILE_WIDTH 16.

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

   Course lecture & textbook.

7. **Optimization 7: *Using Streams to overlap computation with data transfer (4 point)***

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

      I choose the optimization: Using Streams to overlap computation with data transfer. This optimization is a recently learnt one without practicing, so I decided to implement this optimization as a review of the lecture.

   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?

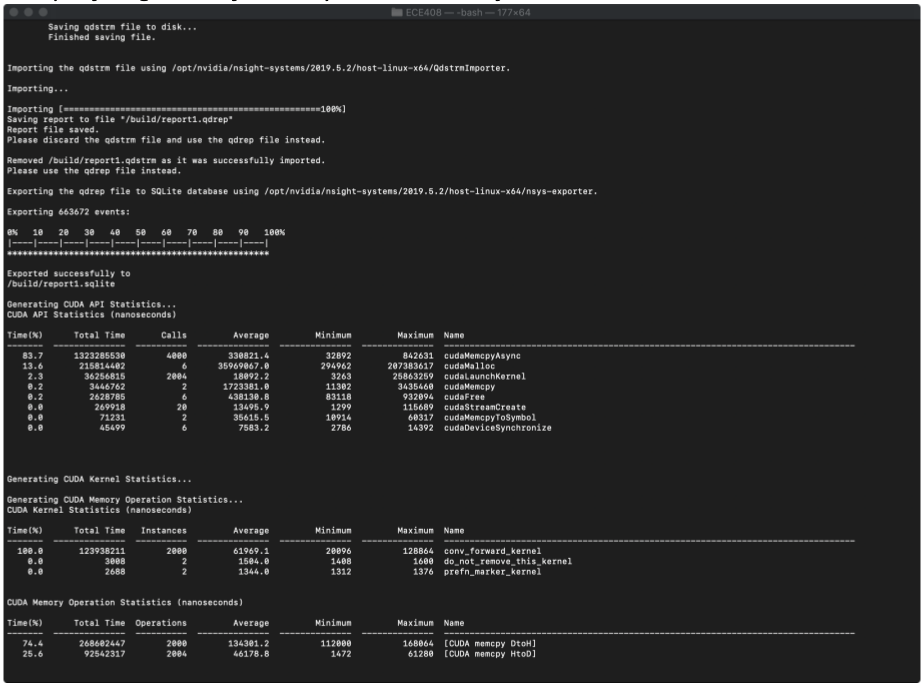
Using multiple streams, we can overlap the data load process with the compute process and speed up, as illustrated in the lecture. initially I thought this optimization would work since the tasks of loading data and computing data are parallelized. However, I found that this optimization actually cannot increase performance when testing.
This optimization synergizes with the previous 2 optimizations: coalesced memory access and 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 10k images using this optimization (including any previous optimizations also used). ***Note that the value in the following table cannot reflect the true op times for this optimization! For this implementation all the work is done in conv_forward_gpu_prolog. The memory copy is done using cudaMemcpyAsync instead of regular cudaMemcpy, so the timing is not referable.***

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.004917ms*	*0.005638ms*	*0m1.169s*	*0.86*
1000	*0.006175ms*	*0.007566ms*	*0m9.989s*	*0.886*
10000	*0.00705ms*	*0.005867ms*	*m39.057s*	*0.8714*

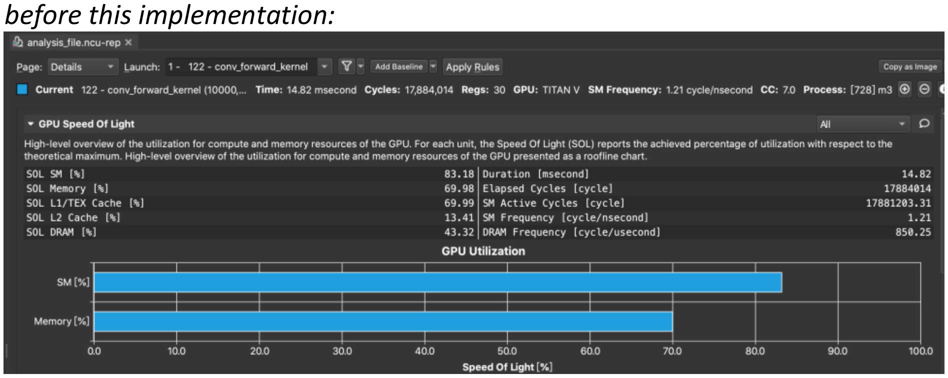
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization successfully improves the performance.
The profiling result from nsys is shown as follows:

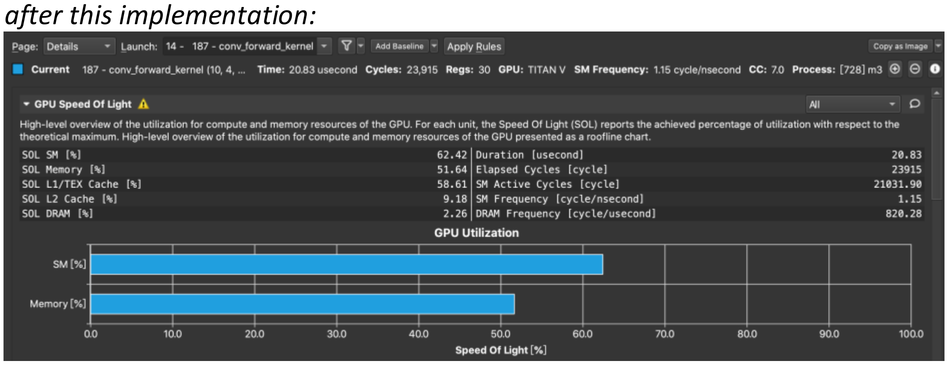


```
ECE408 — -bash — 177×64
       Saving qdstrm file to disk...
       Finished saving file.


Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 663672 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   83.7      1323285530        4000        330821.4           32892          842631  cudaMemcpyAsync
   13.6       215814402           6      35969067.0          294962       207383617  cudaMalloc
    2.3        36256815        2004         18092.2            3263        25863259  cudaLaunchKernel
    0.2         3446762           2       1723381.0           11302         3435460  cudaMemcpy
    0.2         2628785           6        438130.8           83118          932094  cudaFree
    0.0          269918          20         13495.9            1299          115689  cudaStreamCreate
    0.0           71231           2         35615.5           10914           60317  cudaMemcpyToSymbol
    0.0           45499           6          7583.2            2786           14392  cudaDeviceSynchronize



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
  100.0       123938211        2000         61969.1           20096          128864  conv_forward_kernel
    0.0            3008           2          1504.0            1408            1600  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
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   74.4       268602447        2000        134301.2          112000          168064  [CUDA memcpy DtoH]
   25.6        92542317        2004         46178.8            1472           61280  [CUDA memcpy HtoD]
```

As shown, the time for conv_forward_kernel is 123.938ms. Before adding this optimization, the timing result is 73.02ms.
According to the profiling result from Nsight Compute, the GPU Speed of Light is worse compared to the previous "coalesced+kernel value in constant" version before implementing this optimization:
before this implementation:



after this implementation:



While checking the result from Nsight Compute, I noticed the following:

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1000	Registers Per Thread [register/thread]	30
Block Size	256	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	256000	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	1.56	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

Recommendations

Launch Configuration	[Warning] A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full waves and a partial wave of 360 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 30.1%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.

This refers to "wave", something that is not mentioned in the lecture. After checking the code, I realized that I set the SegSize to be 10, since I used 10 streams and the minimum batch size to be test is 100. In order to achieve a faster time for batch size 10000, I modified SegSize to be 1000.
Then it turns out that this modification also improved op time to 72.43ms: (compared to 73.02ms before)

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------
  100.0        72430788          20       3621539.4         1508315         5749353  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1280            1376  prefn_marker_kernel
```

However, the layer time is reduced to less than 1000ms, which is a significant breakthrough for this milestone.

Then I realized that maybe I can adjust the TILE_WIDTH to get a better a better op time. I changed TILE_WIDTH from 16 to 20 according to my optimization 6: Sweeping various parameter to find best values for block sizes. Then the op time is reduced into 66.34ms!

```
ECE408 — -bash — 178×64

Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------
  100.0        66340476          20       3317023.8         1630711         5014402  conv_forward_kernel
    0.0            2944           2          1472.0            1472            1472  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
-------  --------------  ----------  --------------  --------------  --------------  ----------------------
   91.0       904118389          20      45205919.5        37461893        52932842  [CUDA memcpy DtoH]
    9.0        89580531          24       3732522.1            1472         4804996  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  ----------------------
        1722500.0              20            86125.0          72250.000           100000.0  [CUDA memcpy DtoH]
         538919.0              24            22454.0              0.004            28890.0  [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     92913198899         943      98529373.2           23509       100188533  sem_timedwait
   33.3     92869855218         942      98587956.7           34483       100249073  poll
   22.0     61381536540           2   30690768270.0     22536834162     38844702378  pthread_cond_wait
   11.3     31509259628          63     500146978.2       500100030       500166364  pthread_cond_timedwait
    0.0        87002586         907         95923.5            1108        16914818  ioctl
    0.0        19179017          26        737654.5            1222        19108096  fopen
    0.0        17281710        9072          1905.0            1230           18168  read
    0.0         2907839          97         29977.7            1115         1222991  mmap
    0.0         1193100         101         11812.9            4535           31298  open64
    0.0          375023           1        375023.0          375023          375023  pthread_mutex_lock
    0.0          278465           5         55693.0           42441           81702  pthread_create
    0.0          154594           3         51531.3           48433           55450  fgets
    0.0           63911          14          4565.1            1317           11823  munmap
    0.0           62740          15          4182.7            2486            6898  write
    0.0           45350           7          6478.6            3380            8532  fflush
    0.0           27220           3          9073.3            2508           14437  fopen64
    0.0           26545           5          5309.0            2732            7334  open
    0.0           24475          11          2225.0            1027            8543  fclose
    0.0           15759           2          7879.5            4269           11490  pthread_cond_signal
    0.0           13147           2          6573.5            6388            6759  socket
    0.0            7772           1          7772.0            7772            7772  pipe2
    0.0            7275           1          7275.0            7275            7275  connect
    0.0            3180           1          3180.0            3180            3180  fwrite
    0.0            2454           2          1227.0            1115            1339  fcntl
    0.0            1384           1          1384.0            1384            1384  bind
```

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

Course lecture & textbook.

8. Summary:

Implemented optimizations:

Optimization 1: *coalesced memory access*

Optimization 2: *Weight matrix (kernel values) in constant memory (1 point)*

Optimization 3: *Tiled shared memory convolution (2 points)*

Optimization 4: *Input channel reduction: atomics (2 point)*

Optimization 5: *Input channel reduction: tree (3 point)*

Optimization 6: *Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (1 point)*

Optimization 7: *Using Streams to overlap computation with data transfer (4 point)*

Total points: 13

Fastest OP time:

66.5133ms

optimizations for the fastest op time:

coalesced memory access, weight matrix (kernel values) in constant memory, sweeping various parameters to find best values (block sizes, amount of thread coarsening).

ECE408 — -bash — 168×64

```
* Running bash -c "time ./m3 100"   \\ Output will appear after run is complete.
Test batch size: 100
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 9.90366 ms
Op Time: 1.11832 ms
Conv-GPU==
Layer Time: 135.05 ms
Op Time: 1.04847 ms

Test Accuracy: 0.86


real    0m4.221s
user    0m1.092s
sys     0m0.162s
* Running bash -c "time ./m3 1000"   \\ Output will appear after run is complete.
Test batch size: 1000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 64.8801 ms
Op Time: 2.54748 ms
Conv-GPU==
Layer Time: 152.519 ms
Op Time: 5.52425 ms

Test Accuracy: 0.886


real    0m12.218s
user    0m9.821s
sys     0m0.292s
* Running bash -c "time ./m3 10000"   \\ Output will appear after run is complete.
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 738.668 ms
Op Time: 17.3814 ms
Conv-GPU==
Layer Time: 475.379 ms
Op Time: 49.1319 ms

Test Accuracy: 0.8714


real    1m40.505s
user    1m37.001s
sys     0m1.480s
* Running profile --stats=true ./m3   \\ Output will appear after run is complete.
**** collection configuration ****
        force-overwrite = false
        stop-on-exit = true
        export_sqlite = true
        stats = true
        capture-range = none
        stop-on-range-end = false
        Beta: ftrace events:
        ftrace-keep-user-config = false
        trace-GPU-context-switch = false
        delay = 0 seconds
        duration = 0 seconds
```

ECE408 — -bash — 178×64

```
Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.

Importing...

Importing [==================================================100%]
Saving report to file "/build/report1.qdrep"
Report file saved.
Please discard the qdstrm file and use the qdrep file instead.

Removed /build/report1.qdstrm as it was successfully imported.
Please use the qdrep file instead.

Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.

Exporting 649122 events:

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

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

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   79.7      1005106932           6     167517822.0           12275       540789683  cudaMemcpy
   13.2       165913947           6      27652324.5          282055       161189589  cudaMalloc
    5.2        65458812           6      10909802.0            3050        49301263  cudaDeviceSynchronize
    1.7        21789715           6       3631619.2           16332        21676207  cudaLaunchKernel
    0.2         2711658           6        451943.0           84066          910438  cudaFree
    0.0          175514           2         87757.0           83954           91560  cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
  100.0        65432559           2      32716279.5        16133253        49299306  conv_forward_kernel
    0.0            2976           2          1488.0            1408            1568  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1312            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------------------------------------------------------
   92.9       929390762           2     464695381.0       389443232       539947530  [CUDA memcpy DtoH]
    7.1        70865499           6      11810916.5            1472        36019932  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------------------------------------------------------------------
        1722500.0               2           861250.0         722500.000          1000000.0  [CUDA memcpy DtoH]
         538919.0               6            89819.0              0.004           288906.0  [CUDA memcpy HtoD]
```

Fastest Layer time:

998.918ms

optimizations for the fastest op time:

coalesced memory access, weight matrix (kernel values) in constant memory, Using Streams to overlap computation with data transfer

Ranking: (until 2021.12.4 22:00 Beijing Time)

ECE408 — -bash — 178×64

YOU	RANK	FASTEST
	0	435.366ms
	1	447.645ms
	2	448.034ms
	3	448.342ms
	4	450.665ms
	5	465.654ms
	6	468.011ms
	7	477.938ms
	8	494.019ms
	9	495.656ms
	10	497.392ms
	11	498.291ms
	12	506.678ms
	13	542.398ms
	14	614.12ms
	15	647.13ms
	16	656.432ms
	17	786.458ms
	18	973.319ms
	19	984.068ms
	20	994.248ms
	21	994.719ms
zihaoli5 -->	22	998.918ms
	23	999.15ms
	24	1.000412s
	25	1.004793s
	26	1.012176s
	27	1.012195s
	28	1.01295s
	29	1.017091s
	30	1.018344s
	31	1.020498s
	32	1.025965s
	33	1.027031s
	34	1.027458s
	35	1.028351s
	36	1.029195s
	37	1.029753s
	38	1.032485s
	39	1.032526s
	40	1.032945s
	41	1.033195s
	42	1.03339s
	43	1.033806s
	44	1.035241s
	45	1.036707s
	46	1.037027s
	47	1.038088s
	48	1.03954s
	49	1.039659s
	50	1.040927s
	51	1.040978s
	52	1.042294s
	53	1.044518s
	54	1.044867s
	55	1.047005s
	56	1.047482s
	57	1.048645s
	58	1.048783s
	59	1.051381s