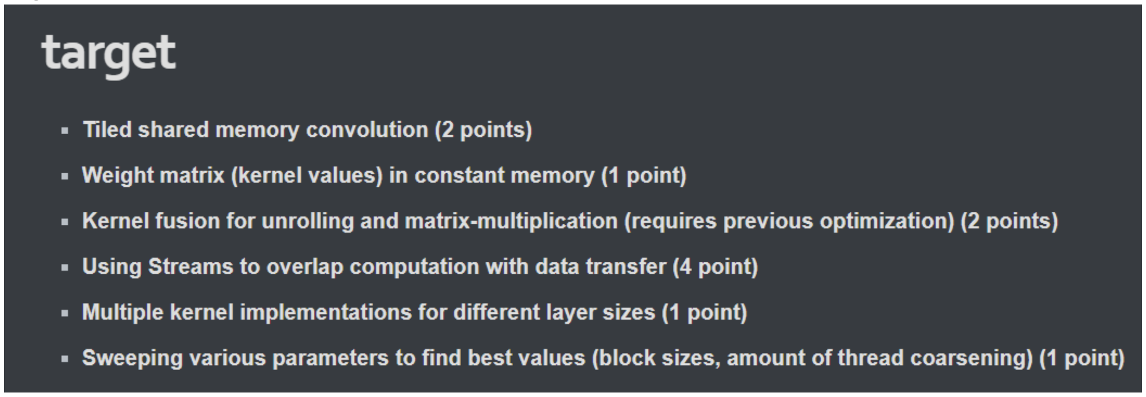
Name: *Song Yifei*
NetID: *yifeis7*
Section: *ZJ1/ZJ2*

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	0.19694 ms	0.567307ms	0m1.113s	*0.86*
1000	2.3186 ms	5.95812 ms	0m9.600s	*0.886*
10000	26.1551 ms	60.231 ms	1m34.921s	*0.8714*

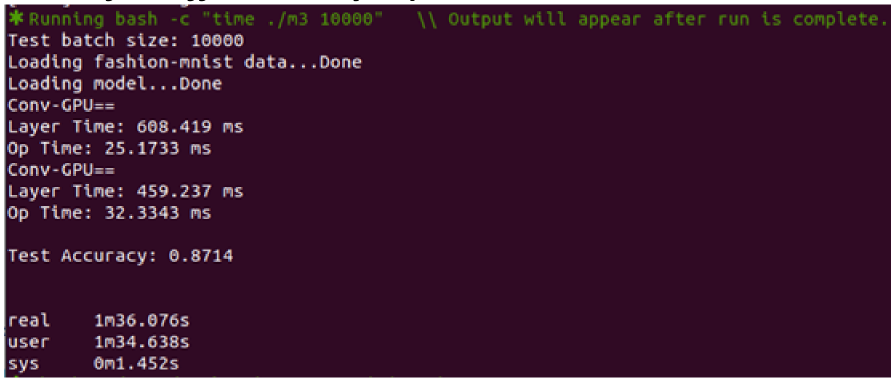
Optimization list:



target

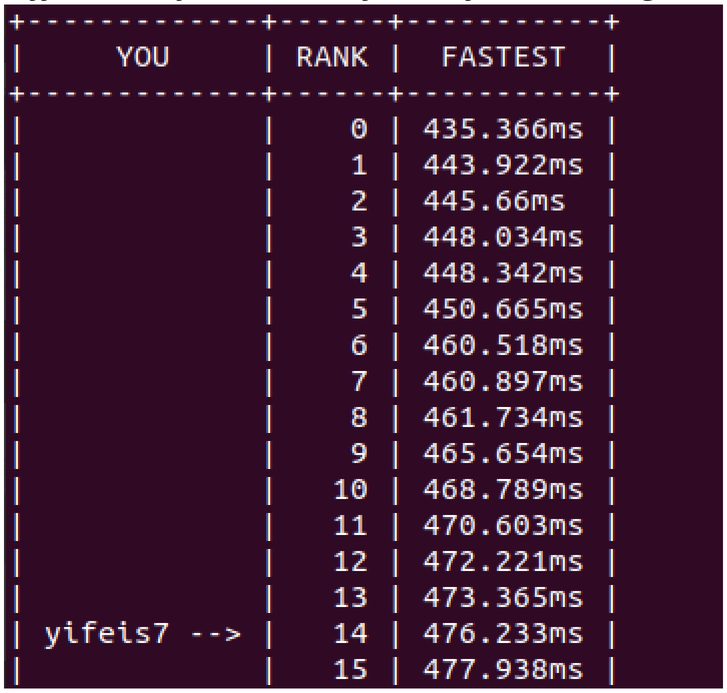
- Tiled shared memory convolution (2 points)
- Weight matrix (kernel values) in constant memory (1 point)
- Kernel fusion for unrolling and matrix-multiplication (requires previous optimization) (2 points)
- Using Streams to overlap computation with data transfer (4 point)
- Multiple kernel implementations for different layer sizes (1 point)
- Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (1 point)

Best op time performance(57.5076 ms):
(multiple kernel for different layer + weight matrix in constant memory+kernel fusion for unrolling and matrix multiplication + multiple kernel for diffenent layer)



```
✱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: 608.419 ms
Op Time: 25.1733 ms
Conv-GPU==
Layer Time: 459.237 ms
Op Time: 32.3343 ms

Test Accuracy: 0.8714


real    1m36.076s
user    1m34.638s
sys     0m1.452s
```

Best layer time performance(476.233 ms):
(overlap between data transfer and computation + multiple kernel for different layer + kernel fusion for unrolling and matrix multiplication)



```
+------------+------+-----------+
|    YOU     | RANK |  FASTEST  |
+------------+------+-----------+
|            |    0 | 435.366ms |
|            |    1 | 443.922ms |
|            |    2 | 445.66ms  |
|            |    3 | 448.034ms |
|            |    4 | 448.342ms |
|            |    5 | 450.665ms |
|            |    6 | 460.518ms |
|            |    7 | 460.897ms |
|            |    8 | 461.734ms |
|            |    9 | 465.654ms |
|            |   10 | 468.789ms |
|            |   11 | 470.603ms |
|            |   12 | 472.221ms |
|            |   13 | 473.365ms |
| yifeis7 -->|   14 | 476.233ms |
|            |   15 | 477.938ms |
```

1. **Optimization 1:** ***Tiled shared memory convolution***

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

      I choose Tiled shared memory convolution because it can take advantage of shared memory and avoid the extra time of reading the global memory. Another reason is that it is popular and was taught many times in the lecture. Also, it is relatively easy to implement.

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

This optimization use divide the data into tiles and store the input and kernel in the shared memory.
Yes, this will increase the performance.
This technique reduces the number of reading and writing from the global memory which will cost longer time than reading and writing from the shared memory. Therefore, the total time for the convolutional layer will decrease.

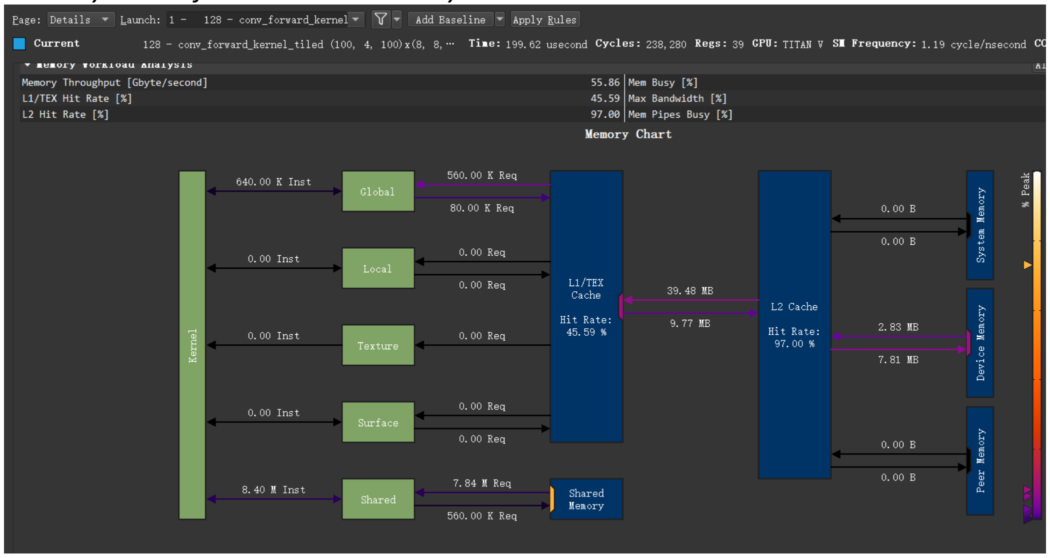
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.210594ms	0.736035ms	0m1.148s	*0.86*
1000	2.43344ms	7.21902ms	0m9.561s	*0.886*
10000	24.9283 ms	72.2975 ms	1m35.655s	*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).

The result was unsuccessful compared with the baseline, but the op time falls between 70-170ms.

Memory chart of Tiled shared memory convolution:



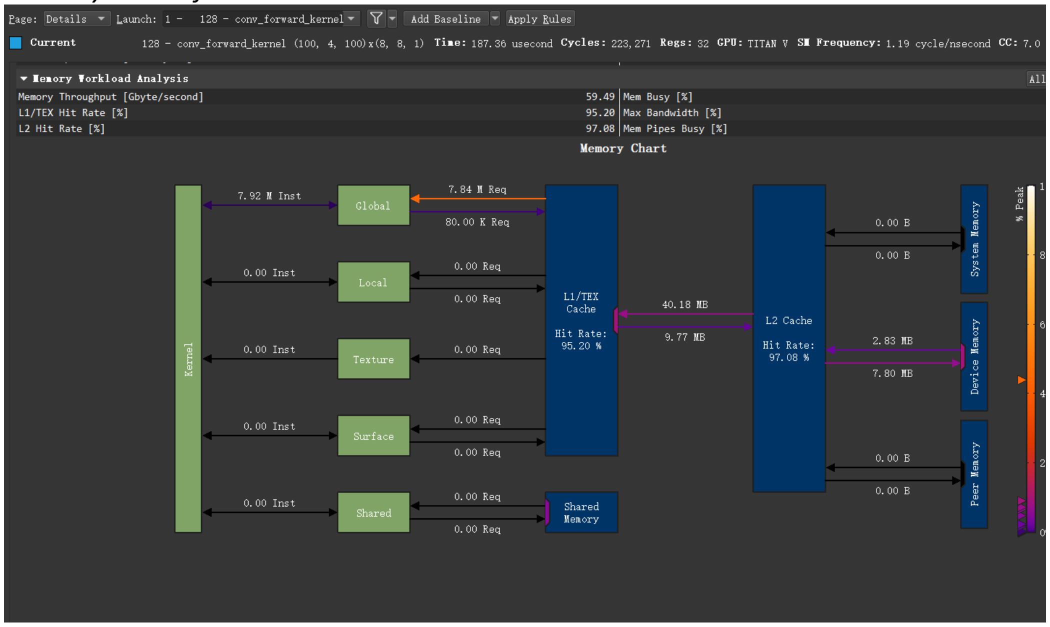
Page: Details | Launch: 1 - 128 - conv_forward_kernel | Add Baseline | Apply Rules

Current 128 - conv_forward_kernel_tiled (100, 4, 100)x(8, 8,… **Time:** 199.62 usecond **Cycles:** 238,280 **Regs:** 39 **GPU:** TITAN V **SM Frequency:** 1.19 cycle/nsecond CC

Memory Workload Analysis

```
Memory Throughput [Gbyte/second]      55.86 | Mem Busy [%]
L1/TEX Hit Rate [%]                   45.59 | Max Bandwidth [%]
L2 Hit Rate [%]                       97.00 | Mem Pipes Busy [%]
```

Memory chart of baseline:



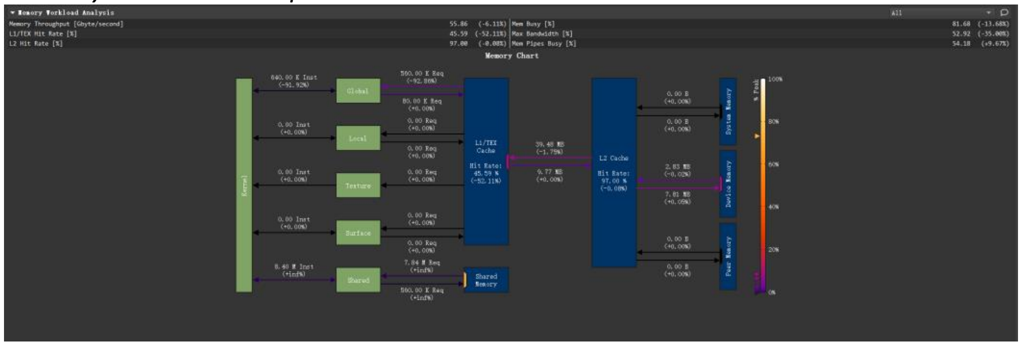
Page: Details | Launch: 1 - 128 - conv_forward_kernel | Add Baseline | Apply Rules

Current 128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1) **Time:** 187.36 usecond **Cycles:** 223,271 **Regs:** 32 **GPU:** TITAN V **SM Frequency:** 1.19 cycle/nsecond **CC:** 7.0

Memory Workload Analysis

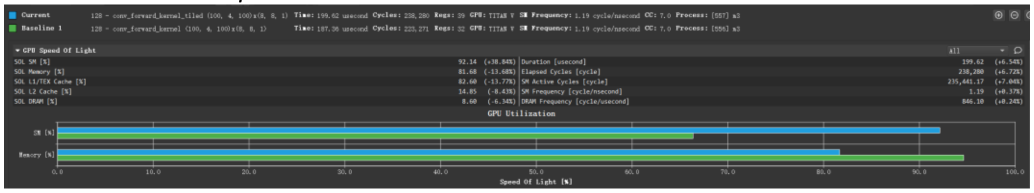
```
Memory Throughput [Gbyte/second]      59.49 | Mem Busy [%]
L1/TEX Hit Rate [%]                   95.20 | Max Bandwidth [%]
L2 Hit Rate [%]                       97.08 | Mem Pipes Busy [%]
```

Memory chart with comparation:



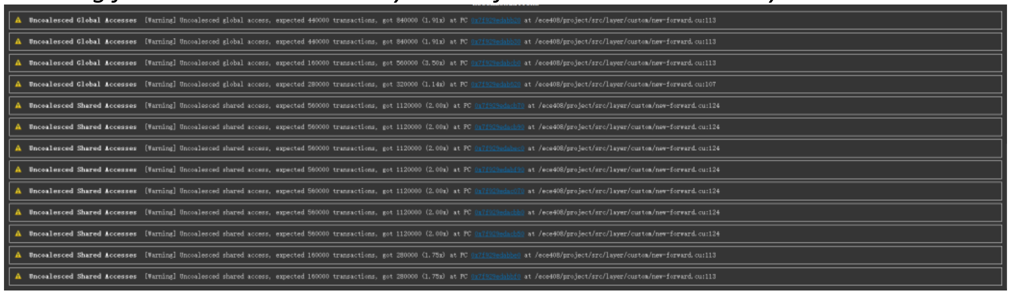
```
▾ Memory Workload Analysis                                                          All
Memory Throughput [Gbyte/second]        55.86  (-6.11%) | Mem Busy [%]              81.68  (-13.68%)
L1/TEX Hit Rate [%]                     45.59 (-52.11%) | Max Bandwidth [%]         52.92  (-35.08%)
L2 Hit Rate [%]                         97.00  (-0.08%) | Mem Pipes Busy [%]        54.18   (+9.67%)
```

GPU utilization comparation:



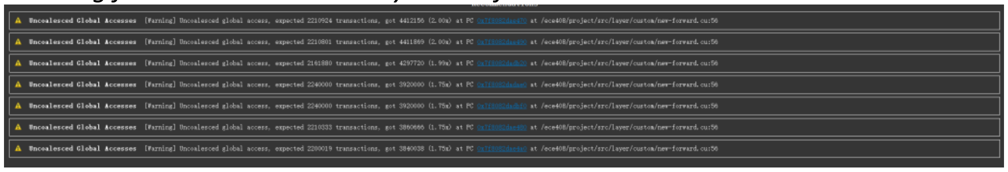
```
■ Current     128 - conv_forward_kernel_tiled (100, 4, 100)x(8, 8, 1) Time: 199.62 usecond Cycles: 238,280 Regs: 39 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [557] m3
■ Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)       Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

▾ GPU Speed Of Light                                                                All
SOL SM [%]               92.14  (+38.84%) | Duration [usecond]                   199.62  (+6.54%)
SOL Memory [%]           81.68  (-13.68%) | Elapsed Cycles [cycle]              238,280  (+6.72%)
SOL L1/TEX Cache [%]     82.60  (-13.77%) | SM Active Cycles [cycle]         235,441.17  (+7.04%)
SOL L2 Cache [%]         14.85   (-8.43%) | SM Frequency [cycle/nsecond]           1.19  (+0.37%)
SOL DRAM [%]              8.60   (-6.34%) | DRAM Frequency [cycle/usecond]       846.10  (+0.24%)
```

According to the memory charts above, we can see that the data throughput between kernel and global memory has decreased significantly, and the throughput between kernel and shared memory increases.
The reason may be that this optimization causes more control divergences than the baseline version when loading data from the global memory into the shared memory.
It can be inferred from the condition of memory coalesce below:
Warning for uncoalesced memory access for tiled shared memory convolution:



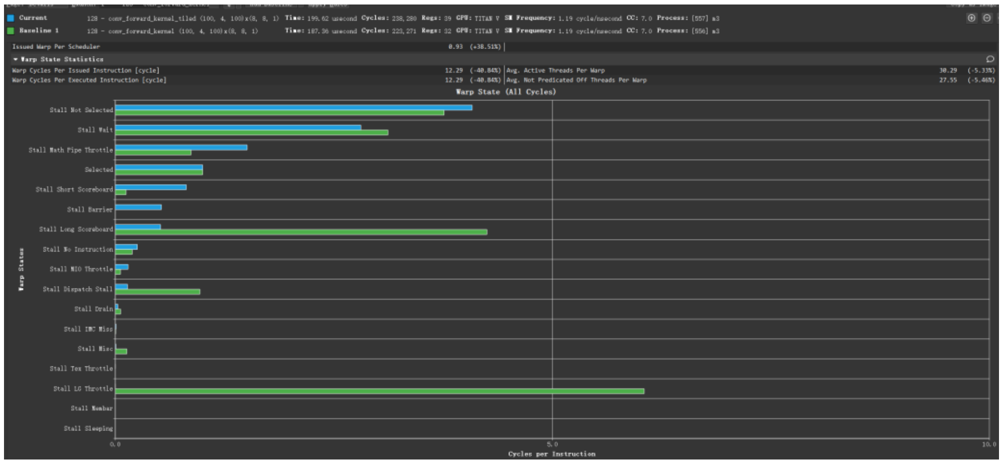
```
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 440000 transactions, got 840000 (1.91x) at PC 0x7f929edabb20 at /ece408/project/src/layer/custom/new-forward.cu:113
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 440000 transactions, got 840000 (1.91x) at PC 0x7f929edabb30 at /ece408/project/src/layer/custom/new-forward.cu:113
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 160000 transactions, got 560000 (3.50x) at PC 0x7f929edabcb0 at /ece408/project/src/layer/custom/new-forward.cu:113
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 280000 transactions, got 320000 (1.14x) at PC 0x7f929edab620 at /ece408/project/src/layer/custom/new-forward.cu:107
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edacb70 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edacb90 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edabac0 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edabf90 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edac070 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edacbb0 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 560000 transactions, got 1120000 (2.00x) at PC 0x7f929edacb50 at /ece408/project/src/layer/custom/new-forward.cu:124
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 160000 transactions, got 280000 (1.75x) at PC 0x7f929edabbe0 at /ece408/project/src/layer/custom/new-forward.cu:113
⚠ Uncoalesced Shared Accesses  [Warning] Uncoalesced shared access, expected 160000 transactions, got 280000 (1.75x) at PC 0x7f929edabbf0 at /ece408/project/src/layer/custom/new-forward.cu:113
```

Warning for uncoalesced memory access for baseline version:



```
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2210924 transactions, got 4412156 (2.00x) at PC 0x7f8082dae470 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2210801 transactions, got 4411869 (2.00x) at PC 0x7f8082dae490 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2161880 transactions, got 4297720 (1.99x) at PC 0x7f8082dadb20 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2240000 transactions, got 3920000 (1.75x) at PC 0x7f8082dadae0 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2240000 transactions, got 3920000 (1.75x) at PC 0x7f8082dadbf0 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2210333 transactions, got 3860666 (1.75x) at PC 0x7f8082dae480 at /ece408/project/src/layer/custom/new-forward.cu:56
⚠ Uncoalesced Global Accesses  [Warning] Uncoalesced global access, expected 2200019 transactions, got 3840038 (1.75x) at PC 0x7f8082dae4a0 at /ece408/project/src/layer/custom/new-forward.cu:56
```

As shown in the above graphs, the memory uncoalesce for tiled shared memory convolution is severer than the baseline version.
Warp state statistics comparation:



```
■ Current     128 - conv_forward_kernel_tiled (100, 4, 100)x(8, 8, 1)  Time: 199.62 usecond  Cycles: 238,280  Regs: 39  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [557] a3
■ Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond  Cycles: 223,271  Regs: 32  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [556] a3
Issued Warp Per Scheduler                          0.93  (+38.51%)
▾ Warp State Statistics
Warp Cycles Per Issued Instruction [cycle]        12.29  (-40.84%)  Avg. Active Threads Per Warp                 30.29  (-5.33%)
Warp Cycles Per Executed Instruction [cycle]      12.29  (-40.84%)  Avg. Not Predicated Off Threads Per Warp     27.55  (-5.46%)
```

As shown above, the number of active threads per warp is also reduced.

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

   I refer to chapter 16 of the textbook when writing this kernel.

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

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

      I choose to use weight matrix (kernel values) in constant memory. This is relatively an easy trick and it can reduce the time of reading convolution kernel to some extent. Another reason is that we are told that the parameters of the training data won't change when the code is tested. This allows us to predefine the size of the array in the constant memory.

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

This optimization works by taking advantage of the constant memory of the GPU. It allows faster reading than global memory. Therefore, the memory usage will decrease to some extent.
Yes, this will increase the performance.
The reason is same as above: This optimization works by taking advantage of the constant memory of the GPU. It allows faster reading than global memory. Therefore, the memory usage will decrease to some extent.
I tried to synergize this optimization with many of other optimizations, such as Tiled shared memory convolution and Shared memory matrix multiplication and input matrix unrolling. Most of the results are satisfying.

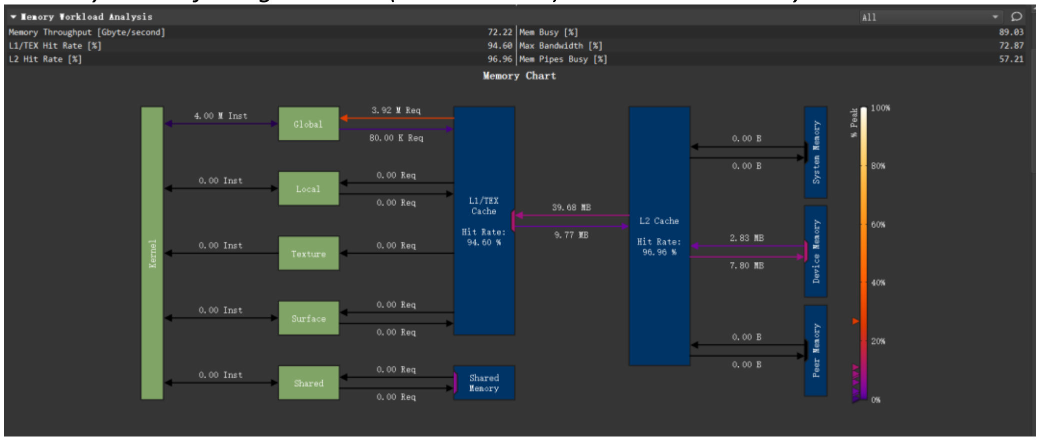
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.168776ms	1.17202 ms	0m2.345s	*0.86*
1000	2.34892ms	6.0269 ms	0m10.394s	*0.886*
10000	30.9806ms	54.0036 ms	1m41.895s	*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 is successful.
The reason is that this optimization leads to less reading of the global memory by transferring the convolution kernel from the global memory to the constant memory.

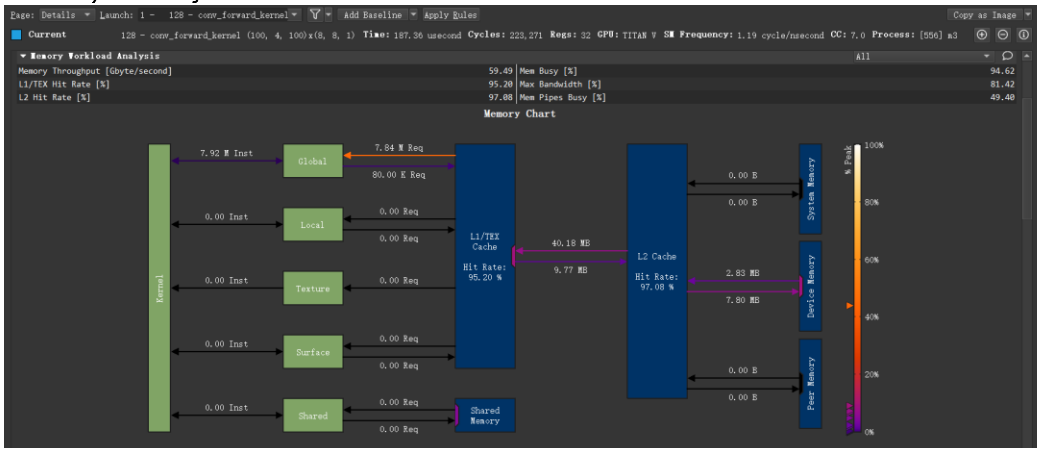
Memory chart of Weight matrix (kernel values) in constant memory:



▾ Memory Workload Analysis

Memory Throughput [Gbyte/second]	72.22	Mem Busy [%]	89.03
L1/TEX Hit Rate [%]	94.60	Max Bandwidth [%]	72.87
L2 Hit Rate [%]	96.96	Mem Pipes Busy [%]	57.21

Memory chart of baseline:



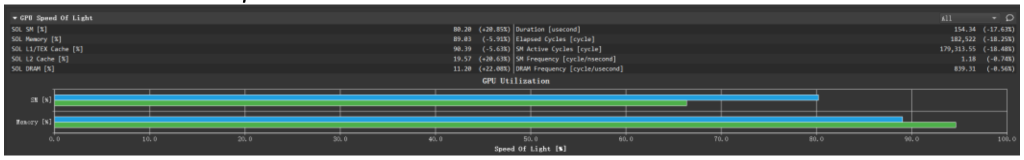
Page: Details Launch: 1 - 128 - conv_forward_kernel Add Baseline Apply Rules Copy as Image

Current 128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1) **Time:** 187.36 usecond **Cycles:** 223,271 **Regs:** 32 **GPU:** TITAN V **SM Frequency:** 1.19 cycle/nsecond **CC:** 7.0 **Process:** [556] m3

▾ Memory Workload Analysis

Memory Throughput [Gbyte/second]	59.49	Mem Busy [%]	94.62
L1/TEX Hit Rate [%]	95.20	Max Bandwidth [%]	81.42
L2 Hit Rate [%]	97.08	Mem Pipes Busy [%]	49.40

Memory chart in comparation view:
According to the graph, we can see that the reading throughput from global memory is significantly decreased because we put the kernel into the constant memory.
Due to the above reason, the utilization of SM is promoted from 66.36% to 80.20%.
GPU Utilization comparation:



▾ GPU Speed Of Light

SOL SM [%]	80.20	(+20.85%)	Duration [usecond]	154.34	(-17.63%)
SOL Memory [%]	89.03	(-5.91%)	Elapsed Cycles [cycle]	182,522	(-18.25%)
SOL L1/TEX Cache [%]	90.39	(-5.63%)	SM Active Cycles [cycle]	179,313.55	(-18.48%)
SOL L2 Cache [%]	19.57	(+20.63%)	SM Frequency [cycle/nsecond]	1.18	(-0.74%)
SOL DRAM [%]	11.20	(+22.08%)	DRAM Frequency [cycle/usecond]	839.31	(-0.56%)

The SM utilization for weight in constant version is significantly promoted, and the amount of memory access is also reduced. This can further prove that this optimization is successful.

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

Yes, I get this idea from the project guideline on github.
https://github.com/Illinois-impact/ece408_project/tree/2021fa

3. **Optimization 3:** ***Kernel fusion for unrolling and matrix-multiplication***

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

      I choose to implement Kernel fusion for unrolling and matrix-multiplication.
      This optimization is based on the Shared memory matrix multiplication and input matrix unrolling. However, the basic version is even slower than the total layer time of the baseline version. After the TA told me that the layer time will decrease after kernel fusion, I tried to adjust the baisic version to the "kernel fusion" version by combining the unrolling process and multiplication process. That is why I implement this 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?

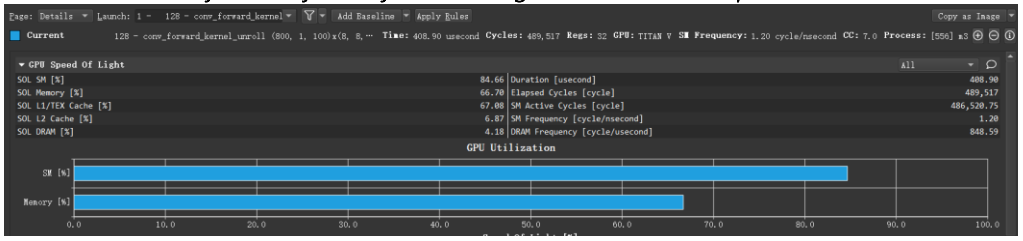
      This optimizaiton is actually the updated version of shared memory multiplication and matrix unroll. It performs matrix unrolling and matrix multiplication at the same time. Essentially, performing matrix multiplication on GPU is much faster. I just take advantage of this feature and implement this optimization.
      Yes, I think it would increase the performance.
      The reason is *matrix multiplication is highly optimized on all hardware platforms. Matrix multiplication is especially fast on GPUs because it has a high ratio of floating point operations per byte of global memory data access. Also, in this implementation I unroll the input matrix and perform basic matrix multiplication in the same kernel. The time wasted in launching kernel is avoided. Also I used shared memory for multiplication which will also save some time compared with the baseline.*
      No, it is implemented based on the baseline version.

   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	0.419688ms	0.310472ms	0m1.207s	*0.86*
1000	5.04392 ms	3.10896 ms	0m10.287s	*0.886*
10000	53.5815 ms	32.6401 ms	1m40.993s	*0.8714*

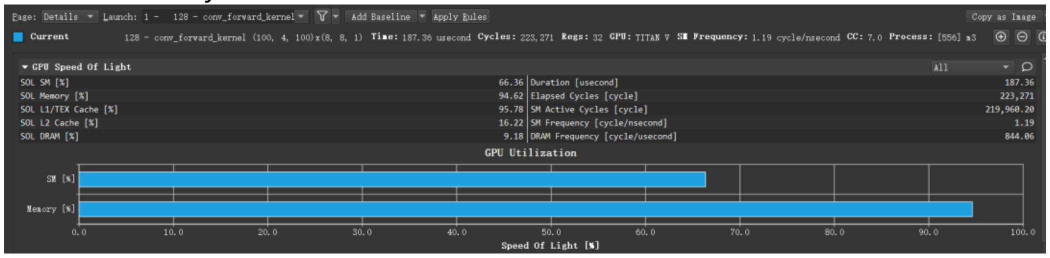
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).
Yes, considering the reduction of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

GPU Utilization of Kernel fusion for unrolling and matrix-multiplication:



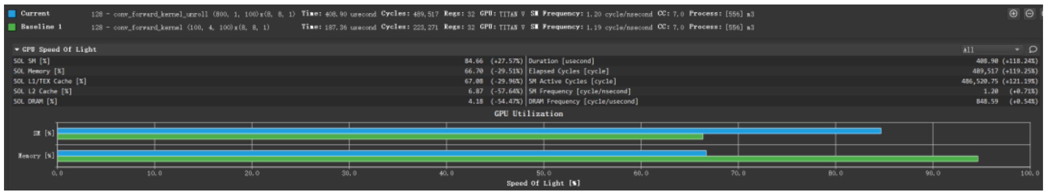
```
Page: Details  Launch: 1 - 128 - conv_forward_kernel  Add Baseline  Apply Rules  Copy as Image
Current   128 - conv_forward_kernel_unroll (800, 1, 100)x(8, 8,…  Time: 408.90 usecond  Cycles: 489,517  Regs: 32  GPU: TITAN V  SM Frequency: 1.20 cycle/nsecond  CC: 7.0  Process: [556] m3

GPU Speed Of Light                                                              All
SOL SM [%]              84.66 | Duration [usecond]                    408.90
SOL Memory [%]          66.70 | Elapsed Cycles [cycle]                489,517
SOL L1/TEX Cache [%]    67.08 | SM Active Cycles [cycle]              486,520.75
SOL L2 Cache [%]         6.87 | SM Frequency [cycle/nsecond]          1.20
SOL DRAM [%]             4.18 | DRAM Frequency [cycle/usecond]        848.59
```

GPU Utilization of baseline:



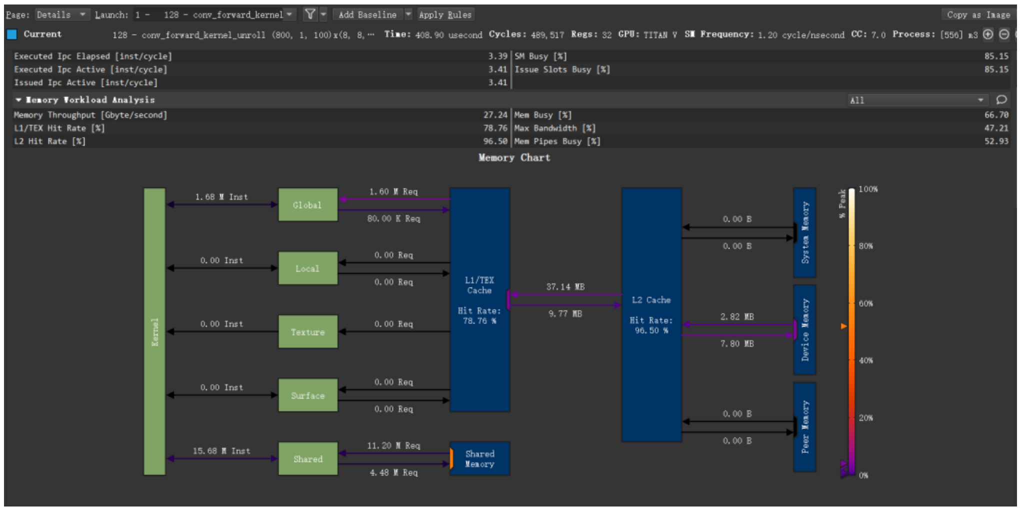
```
Page: Details  Launch: 1 - 128 - conv_forward_kernel  Add Baseline  Apply Rules  Copy as Image
Current   128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)  Time: 187.36 usecond  Cycles: 223,271  Regs: 32  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [556] m3

GPU Speed Of Light                                                              All
SOL SM [%]              66.36 | Duration [usecond]                    187.36
SOL Memory [%]          94.62 | Elapsed Cycles [cycle]                223,271
SOL L1/TEX Cache [%]    95.78 | SM Active Cycles [cycle]              219,960.20
SOL L2 Cache [%]        16.22 | SM Frequency [cycle/nsecond]          1.19
SOL DRAM [%]             9.18 | DRAM Frequency [cycle/usecond]        844.06
```

GPU utilization comparation:



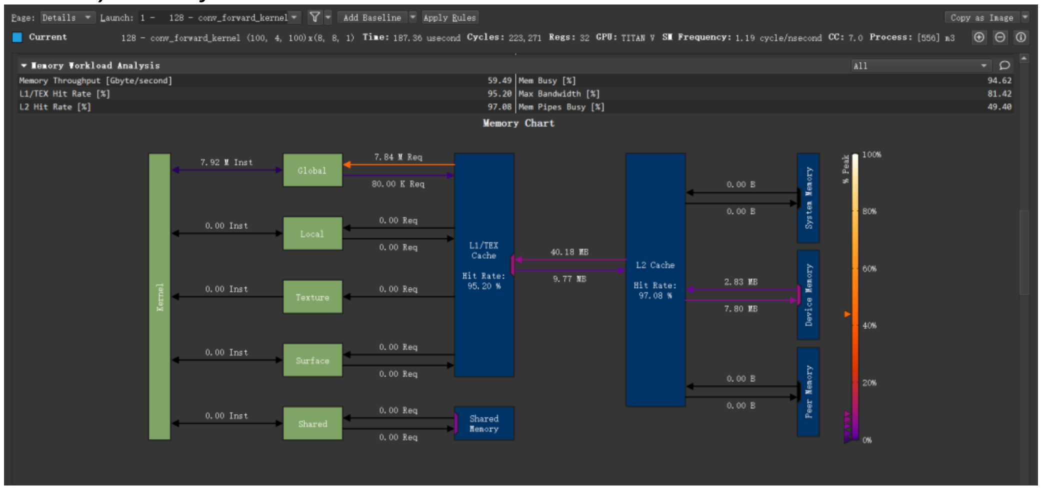
```
Current     128 - conv_forward_kernel_unroll (800, 1, 100)x(8, 8, 1)  Time: 408.90 usecond  Cycles: 489,517  Regs: 32  GPU: TITAN V  SM Frequency: 1.20 cycle/nsecond  CC: 7.0  Process: [556] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)         Time: 187.36 usecond  Cycles: 223,271  Regs: 32  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [556] m3

GPU Speed Of Light                                                                         All
SOL SM [%]              84.66  (+27.57%) | Duration [usecond]               408.90  (+118.24%)
SOL Memory [%]          66.70  (-29.51%) | Elapsed Cycles [cycle]           489,517 (+119.25%)
SOL L1/TEX Cache [%]    67.08  (-29.96%) | SM Active Cycles [cycle]      486,520.75 (+121.19%)
SOL L2 Cache [%]         6.87  (-57.64%) | SM Frequency [cycle/nsecond]     1.20    (+0.71%)
SOL DRAM [%]             4.18  (-54.47%) | DRAM Frequency [cycle/usecond]   848.59  (+0.54%)
```

From the above two pictures, the optimization is obvious. The GPU utilization of SM has increased from 66.36% to 84.66%. And the total memory usage SOL has changed from 94.62% to 66.70%. That is quite a leap for the total performance.
Matrix multiplication is highly optimized on all hardware platforms. By changing the convolution process into matrix multiplication, we can promote the utilization of streaming multiprocessors as shown above.
Memory chart of Kernel fusion for unrolling and matrix-multiplication:



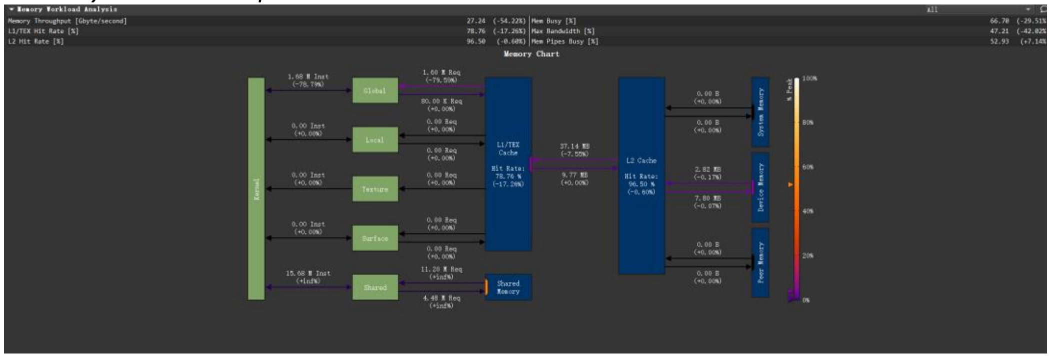
```
Page: Details  Launch: 1 -  128 - conv_forward_kernel  Add Baseline  Apply Rules                Copy as Image
Current   128 - conv_forward_kernel_unroll (800, 1, 100)x(8, 8,… Time: 408.90 usecond Cycles: 489,517 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [556] m3
Executed Ipc Elapsed [inst/cycle]     3.39 | SM Busy [%]            85.15
Executed Ipc Active [inst/cycle]      3.41 | Issue Slots Busy [%]   85.15
Issued Ipc Active [inst/cycle]        3.41 |
▼ Memory Workload Analysis                                          All
Memory Throughput [Gbyte/second]     27.24 | Mem Busy [%]           66.70
L1/TEX Hit Rate [%]                  78.76 | Max Bandwidth [%]      47.21
L2 Hit Rate [%]                      96.50 | Mem Pipes Busy [%]     52.93
```

Memory chart of baseline:



```
Page: Details  Launch: 1 -  128 - conv_forward_kernel  Add Baseline  Apply Rules                Copy as Image
Current   128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1) Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3
▼ Memory Workload Analysis                                          All
Memory Throughput [Gbyte/second]     59.49 | Mem Busy [%]           94.62
L1/TEX Hit Rate [%]                  95.20 | Max Bandwidth [%]      81.42
L2 Hit Rate [%]                      97.08 | Mem Pipes Busy [%]     49.40
```

Memory chart in comparation view:



```
▼ Memory Workload Analysis                                                 All
Memory Throughput [Gbyte/second]   27.24  (-54.22%) | Mem Busy [%]         66.70  (-29.51%)
L1/TEX Hit Rate [%]                78.76  (-17.26%) | Max Bandwidth [%]    47.21  (-42.02%)
L2 Hit Rate [%]                    96.50  (-0.60%)  | Mem Pipes Busy [%]   52.93  (+7.14%)
```

According to the above three memory charts, we can also find that the global memory throughput has decreased significantly(-79.5%) after this optimization and the usage of shared memory has increased. This leads to the further reduction of running time.

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

I refered to chapter 16 of the textbook(the part about input matrix unrolling).

4. **Optimization 4: *Using Streams to overlap computation with data transfer***

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

   I choose to implement Using Streams to overlap computation with data transfer. The reason is that it can significantly reduce the total layer time by taking advantage of cuda streams.

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

   This optimization uses the technique which is called device overlap. Simply put, It can simultaneously execute a kernel while performing a copy between device and host memory. Consequently, the extent of parallelism increases.
   Yes, I think it will increase the performance.
   The reason is that this implementation optimizes data transfers between the host and device. The usage of CUDA Streams and Asynchronous Memcpy enables the host code to become more efficient.
   Yes, it synergizes with the Kernel fusion for unrolling and matrix multiplication.

   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).

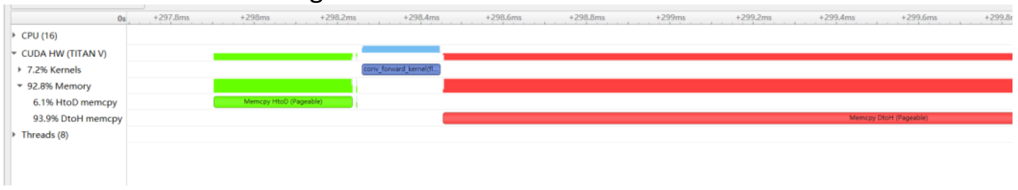
Batch Size	Op Time 1	Op Time 2	Layer time 1	Layer time 2	Total Execution Time	Layer time 1(before optimization)	Layer time 2(before optimization)	Accuracy
100	0.007453 ms	0.004579 ms	6.65024 ms	5.08021 ms	0m1.071s	7.21485 ms	5.67959 ms	*0.86*
1000	0.004706 ms	0.004922 ms	61.4363 ms	46.5194 ms	0m9.143s	67.0995 ms	51.6899 ms	*0.886*
10000	0.00573 ms	0.006463 ms	595.836 ms	473.327 ms	1m32.036s	664.037 ms	488.232 ms	*0.8714*

Note: The op time becomes unavailable when using cuda streams, because I transferred all the data to conv_forward_gpu_prolog, but the layer time do decrease.

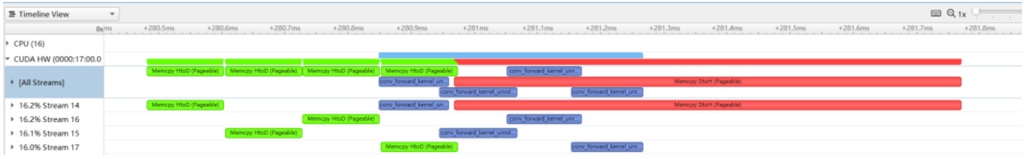
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, it is successful. Though the op time is no longer available, it is obvious that the layer time is reduced.
Before optimization, the kernel copies all the data and then compute all the convolution calculation, then copy all the output back to host. This is a process with low efficiency. By using streams we can create overlap between computing and data transfer which will significantly improve the efficiency and reduce total layer time.

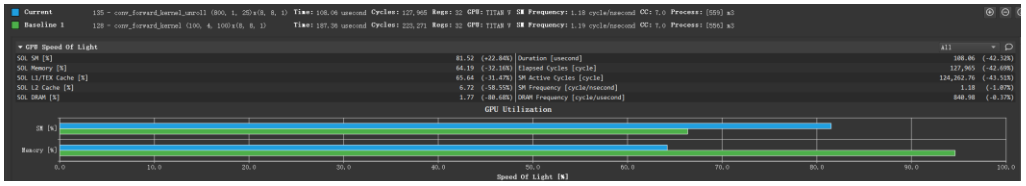
Timeline view before using streams:



Timeline view after using 4 streams(SegSize = 25):

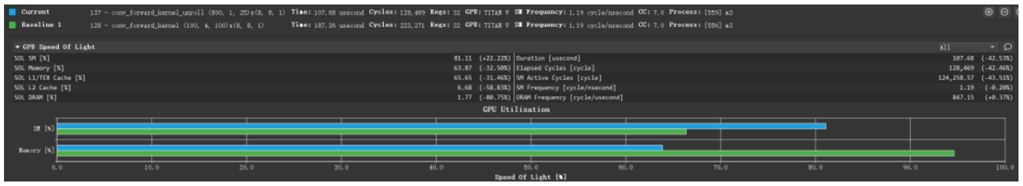


GPU utilization charts for four streams of convolution kernel:
1:



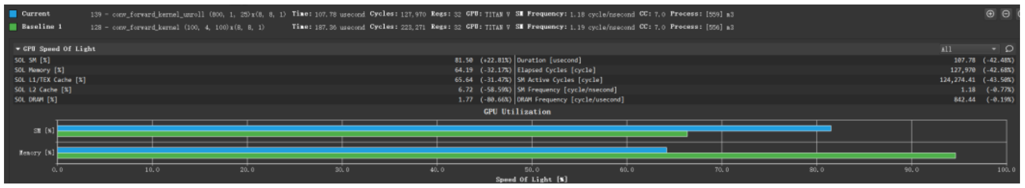
```
Current     135 - conv_forward_kernel_unroll (800, 1, 25)x(8, 8, 1) Time: 108.06 usecond Cycles: 127,965 Regs: 32 GPU: TITAN V SM Frequency: 1.18 cycle/nsecond CC: 7.0 Process: [559] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light
SOL SM [%]             81.52 (+22.84%) | Duration [usecond]               108.06 (-42.32%)
SOL Memory [%]         64.19 (-32.16%) | Elapsed Cycles [cycle]          127,965 (-42.69%)
SOL L1/TEX Cache [%]   65.64 (-31.47%) | SM Active Cycles [cycle]     124,262.76 (-43.51%)
SOL L2 Cache [%]        6.72 (-58.55%) | SM Frequency [cycle/nsecond]       1.18 (-1.07%)
SOL DRAM [%]            1.77 (-80.68%) | DRAM Frequency [cycle/usecond]   840.98 (-0.37%)
```

2:



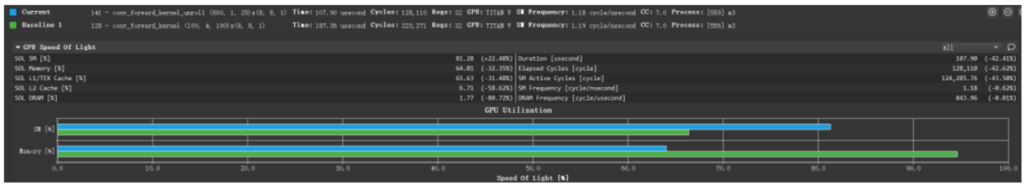
```
Current     137 - conv_forward_kernel_unroll (800, 1, 25)x(8, 8, 1) Time: 107.68 usecond Cycles: 128,469 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [559] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light
SOL SM [%]             81.11 (+22.22%) | Duration [usecond]               107.68 (-42.53%)
SOL Memory [%]         63.87 (-32.50%) | Elapsed Cycles [cycle]          128,469 (-42.46%)
SOL L1/TEX Cache [%]   65.65 (-31.46%) | SM Active Cycles [cycle]     124,258.57 (-43.51%)
SOL L2 Cache [%]        6.68 (-58.83%) | SM Frequency [cycle/nsecond]       1.19 (-0.20%)
SOL DRAM [%]            1.77 (-80.75%) | DRAM Frequency [cycle/usecond]   847.15 (+0.37%)
```

3:



```
Current     139 - conv_forward_kernel_unroll (800, 1, 25)x(8, 8, 1) Time: 107.78 usecond Cycles: 127,970 Regs: 32 GPU: TITAN V SM Frequency: 1.18 cycle/nsecond CC: 7.0 Process: [559] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light
SOL SM [%]             81.50 (+22.81%) | Duration [usecond]               107.78 (-42.48%)
SOL Memory [%]         64.19 (-32.17%) | Elapsed Cycles [cycle]          127,970 (-42.68%)
SOL L1/TEX Cache [%]   65.64 (-31.47%) | SM Active Cycles [cycle]     124,274.41 (-43.50%)
SOL L2 Cache [%]        6.72 (-58.59%) | SM Frequency [cycle/nsecond]       1.18 (-0.77%)
SOL DRAM [%]            1.77 (-80.66%) | DRAM Frequency [cycle/usecond]   842.44 (-0.19%)
```

4:



```
Current     141 - conv_forward_kernel_unroll (800, 1, 25)x(8, 8, 1) Time: 107.90 usecond Cycles: 128,110 Regs: 32 GPU: TITAN V SM Frequency: 1.18 cycle/nsecond CC: 7.0 Process: [559] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light
SOL SM [%]             81.28 (+22.48%) | Duration [usecond]               107.90 (-42.41%)
SOL Memory [%]         64.01 (-32.35%) | Elapsed Cycles [cycle]          128,110 (-42.62%)
SOL L1/TEX Cache [%]   65.63 (-31.48%) | SM Active Cycles [cycle]     124,285.76 (-43.50%)
SOL L2 Cache [%]        6.71 (-58.62%) | SM Frequency [cycle/nsecond]       1.18 (-0.62%)
SOL DRAM [%]            1.77 (-80.72%) | DRAM Frequency [cycle/usecond]   843.96 (-0.01%)
```

According to the four GPU utilization views, we can infer that the performance of four streams all outstand the baseline version. With the overlap between data transfer and computation, the performance is bound to be promoted to some extent.

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

   I referred to the slide of lec 22: GPU-Data-Transfer-vk-FL21.

5. **Optimization 5: *Multiple kernel implementations for different layer sizes***

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

      I choose to implement Multiple kernel implementations for different layer sizes. The reason is that the parameter for those two convolutional layers is different, so we can choose the best version of implementation for each of the layer. Also, this optimization can be synergized with other optimizations easily.

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 by writing separate kernels for different layers. Choosing the best conv_forward kernel for each of layer will result in best total op time.
   Yes, it will increase the performance.
   The reason is that using a single kernel for both layers may not be the best solution because the two convolution layers have different sizes. To reach the maximum performance for both of them, two different kernels are needed. And the performance will be further optimized.
   Yes, it synergizes with shared memory tiled convolution and kernel fusion for unrolling and matrix-multiplication.

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	0.196588ms	0.512293ms	0m1.197s	*0.86*
1000	2.77878ms	4.88471 ms	0m11.128s	*0.886*
10000	29.0417 ms	48.5915 ms	1m49.777s	*0.8714*

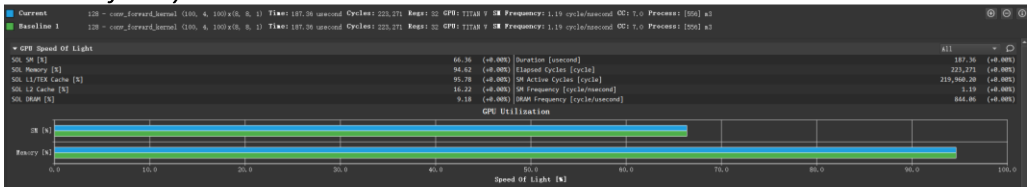
(baseline for layer1; optimization 3 for layer 2)

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 is successful in improving performance. The reason is that the baseline version performs well in op time for layer 1. Though optimization 3 decreases the op time for layer 2 decently, it increases the layer time for layer 1. Therefore, I chose to combine those methods into one implementation. The performance is further optimized.

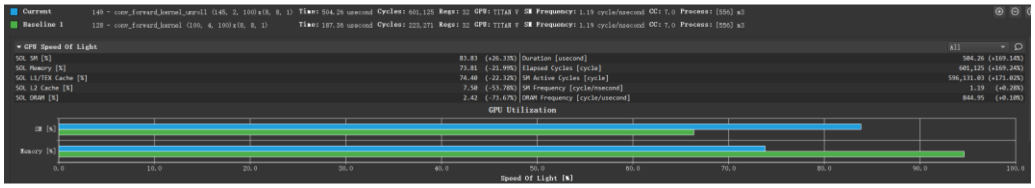
   The baseline kernel is not changed so I only compare the optimization of layer 2 by Nsight-compute.

   Baseline for layer2:



```
Current     128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1) Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1) Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light
SOL SM [%]             66.36  (+0.00%)  Duration [usecond]               187.36  (+0.00%)
SOL Memory [%]         94.62  (+0.00%)  Elapsed Cycles [cycle]          223,271  (+0.00%)
SOL L1/TEX Cache [%]   95.78  (+0.00%)  SM Active Cycles [cycle]     219,960.20  (+0.00%)
SOL L2 Cache [%]       16.22  (+0.00%)  SM Frequency [cycle/nsecond]       1.19  (+0.00%)
SOL DRAM [%]            9.18  (+0.00%)  DRAM Frequency [cycle/usecond]   844.06  (+0.00%)
```

   Kernel fusion for unrolling and matrix-multiplication for layer 2:



```
Current     149 - conv_forward_kernel_unroll (145, 2, 100)x(8, 8, 1) Time: 504.26 usecond Cycles: 601,125 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)        Time: 187.36 usecond Cycles: 223,271 Regs: 32 GPU: TITAN V SM Frequency: 1.19 cycle/nsecond CC: 7.0 Process: [556] m3

▾ GPU Speed Of Light                                                                                     All
SOL SM [%]              83.83  (+26.33%) | Duration [usecond]                   504.26 (+169.14%)
SOL Memory [%]          73.81  (-21.99%) | Elapsed Cycles [cycle]               601,125 (+169.24%)
SOL L1/TEX Cache [%]    74.40  (-22.32%) | SM Active Cycles [cycle]          596,131.03 (+171.02%)
SOL L2 Cache [%]         7.50  (-53.78%) | SM Frequency [cycle/nsecond]            1.19   (+0.28%)
SOL DRAM [%]             2.42  (-73.67%) | DRAM Frequency [cycle/usecond]        844.95   (+0.10%)
```

The improvement of the second layer is significant by checking the GPU utilization chart. This leads to better layer time for layer 2.

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

   No, I did not use any references.

6. **Optimization 6: *Sweeping various parameters***

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

      I choose to implement Sweeping various parameters. The reason is that this is a easy way to optimize the performance of other optimizations whose performance varies significantly when parameters are changed, such as kernel fusion for matrix unrolling and multiplication and shared memory tiled 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?

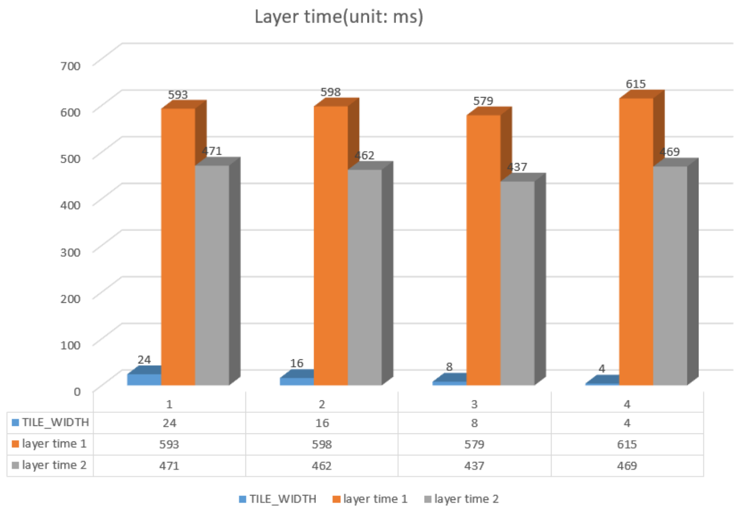
This optimization is done by adjusting the parameters, such as tile_width, block_size, number of cuda streams, etc. The target is to find the global minimum of the running time which also means best performance.
Yes, this will increase the performance.
The reason is this method can change parameter for better performance. In the process of testing multiple combinations of parameters, we may find a better one compared with the baseline.
Yes, it synergizes with Kernel fusion for unrolling and matrix-multiplication and using Streams to overlap computation with data transfer.

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	0.007453ms	0.004579ms	0m1.071s	*0.86*
1000	0.004706ms	0.004922ms	0m9.143s	*0.886*
10000	0.00573ms	0.006463ms	1m32.036s	*0.8714*

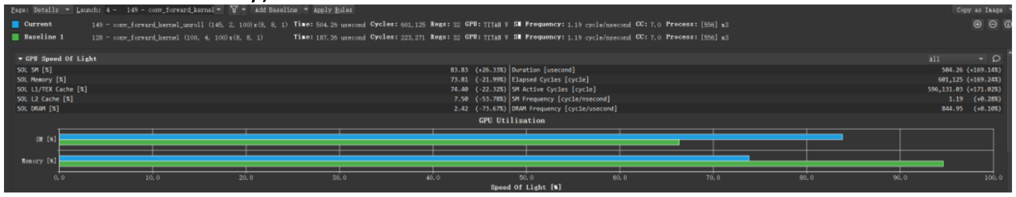
Note: For this optimization, I use streams to overlap computation and data transfer, so the op time is no longer available. I put the results of layer time in the following part.

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 implementation is successful.
I sweep the TILE_WIDTH based on the optimization *Kernel fusion for unrolling and matrix-multiplication and using Streams to overlap computation with data transfer.*
The result is shown below.



	1	2	3	4
TILE_WIDTH	24	16	8	4
layer time 1	593	598	579	615
layer time 2	471	462	437	469

According to my records, the TILE_WIDTH which is equal to 8 outstands all the other parameters based on the previous optimization 3 and optimization 4.
The resource utilization data with TILE_WIDTH = 8 is shown below. It shows better performance compared with the baseline version(higher utilization of SM and lower utilization of memory).



```
Current     149 - conv_forward_kernel_unroll (145, 2, 100)x(8, 8, 1)  Time: 504.26 usecond  Cycles: 601,125  Regs: 32  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [556] m3
Baseline 1  128 - conv_forward_kernel (100, 4, 100)x(8, 8, 1)         Time: 187.36 usecond  Cycles: 223,271  Regs: 32  GPU: TITAN V  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [556] m3

GPU Speed Of Light
SOL SM [%]              83.83  (+26.33%) | Duration [usecond]                    504.26 (+169.14%)
SOL Memory [%]          73.81  (-21.99%) | Elapsed Cycles [cycle]               601,125 (+169.24%)
SOL L1/TEX Cache [%]    74.40  (-22.32%) | SM Active Cycles [cycle]          596,131.03 (+171.02%)
SOL L2 Cache [%]         7.50  (-53.78%) | SM Frequency [cycle/nsecond]            1.19   (+0.28%)
SOL DRAM [%]             2.42  (-73.67%) | DRAM Frequency [cycle/usecond]        844.95   (+0.10%)
```

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

No, I did not use any references.