Name: *Hao Ren*
NetID: *haor2*
Section: *ZJ1*

ECE 408/CS483 Milestone 3 Report

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.246515 ms*	*0.83401 ms*	*0m1.542s*	*0.86*
1000	*2.24226 ms*	*15.3607 ms*	*0m10.669s*	*0.886*
5000	*11.5922 ms*	*40.737 ms*	*0m0.740s*	*0.871*

1. **Optimization 1:** IMPL_INPUT_UNROLLING **Shared memory matrix multiplication and input matrix unrolling (**3 points**)**
   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      The current access pattern and techniques are convolution which isn't optimized and doesn't have special hardware support. To further optimize the performance, we can first convert the whole computing pattern to matrix multiplication, hence a conv2mul kernel is needed to unroll the input matrix.

      The expected improvement might not be that much (or degrades the performance), but it leaves room for off-load the workload to tensor core.

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

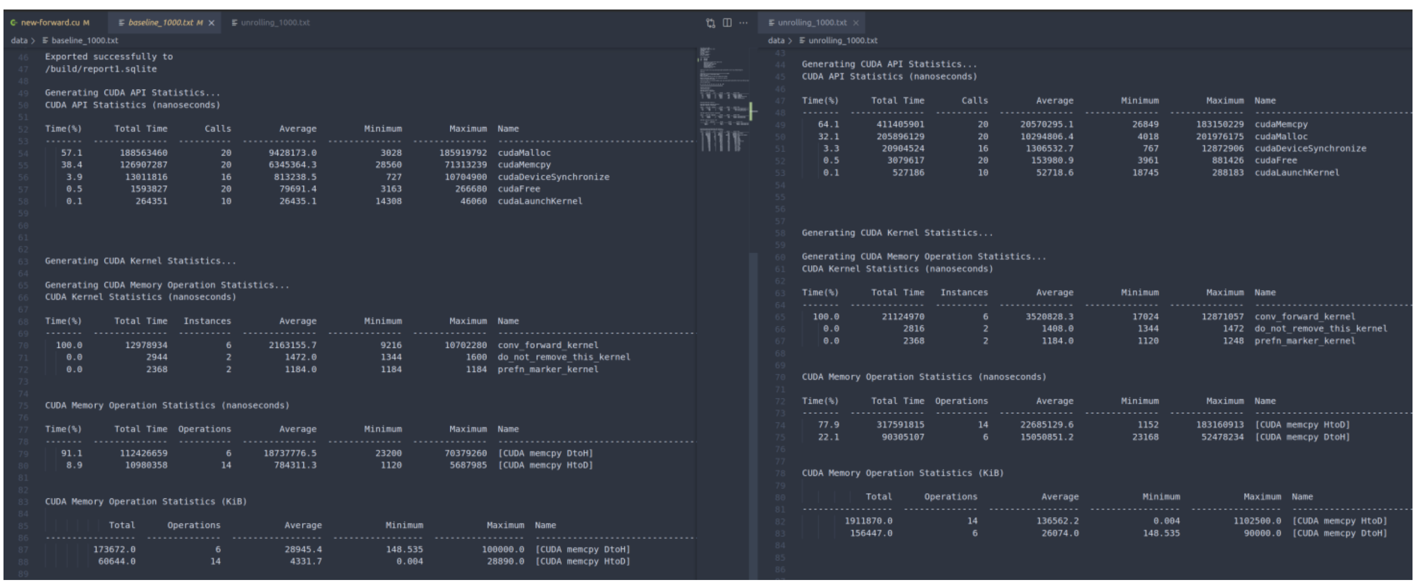
The optimization works by changing the computing method from convolution to matrix multiplication. NO, I don't think it will improve the performance at this point since the unrolling can be expensive at this point and memory coalescing also happens in baseline when set properly. Since this is the first optimization, I will talk about the synergize in next optmization.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.45413 ms*	*0.939222 ms*	*0m0.297s*	*0.86*
1000	*14.3275 ms*	*9.03383 ms*	*0m1.064s*	*0.886*
5000	*71.0609 ms*	*44.6977 ms*	*0m3.832s*	*0.871*

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

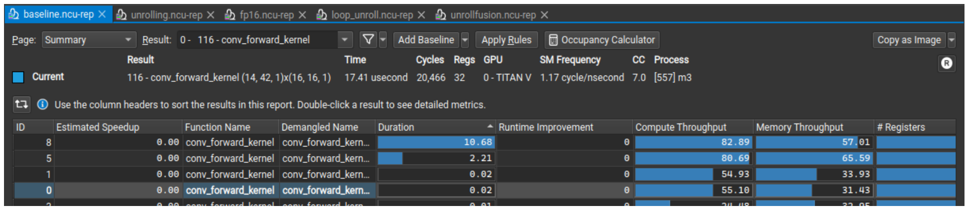
NSYS

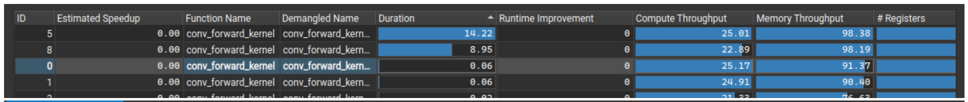


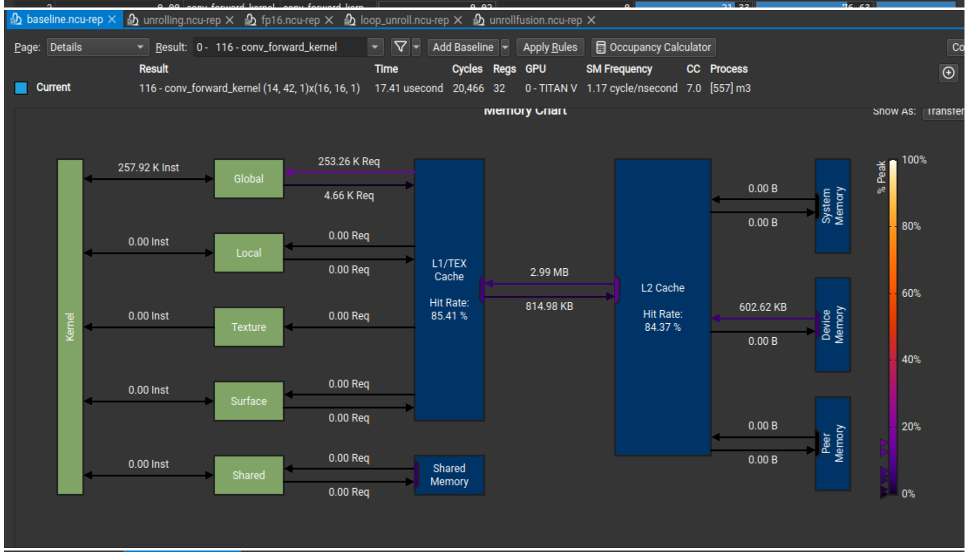

```
Exported successfully to
/build/report1.sqlite

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

Time(%)    Total Time     Calls      Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
   57.1      188563460         20      9428173.0         3028    185919792  cudaMalloc
   38.4      126907287         20      6345364.3        28560     71313239  cudaMemcpy
    3.9       13011816         16       813238.5          727     10704900  cudaDeviceSynchronize
    0.5        1593827         20        79691.4         3163       266680  cudaFree
    0.1         264351         10        26435.1        14308        46060  cudaLaunchKernel




Generating CUDA Kernel Statistics...

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

Time(%)    Total Time   Instances    Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
  100.0       12978934          6      2163155.7         9216     10702280  conv_forward_kernel
    0.0           2944          2         1472.0         1344         1600  do_not_remove_this_kernel
    0.0           2368          2         1184.0         1184         1184  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)    Total Time  Operations    Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
   91.1      112426659          6     18737776.5        23200     70379260  [CUDA memcpy DtoH]
    8.9       10980358         14       784311.3         1120      5687985  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

          Total      Operations       Average          Minimum          Maximum    Name
---------------  ------------  ---------------  ---------------  ---------------  -------------------
       173672.0             6          28945.4          148.535         100000.0  [CUDA memcpy DtoH]
        60644.0            14           4331.7            0.004          28890.0  [CUDA memcpy HtoD]
```

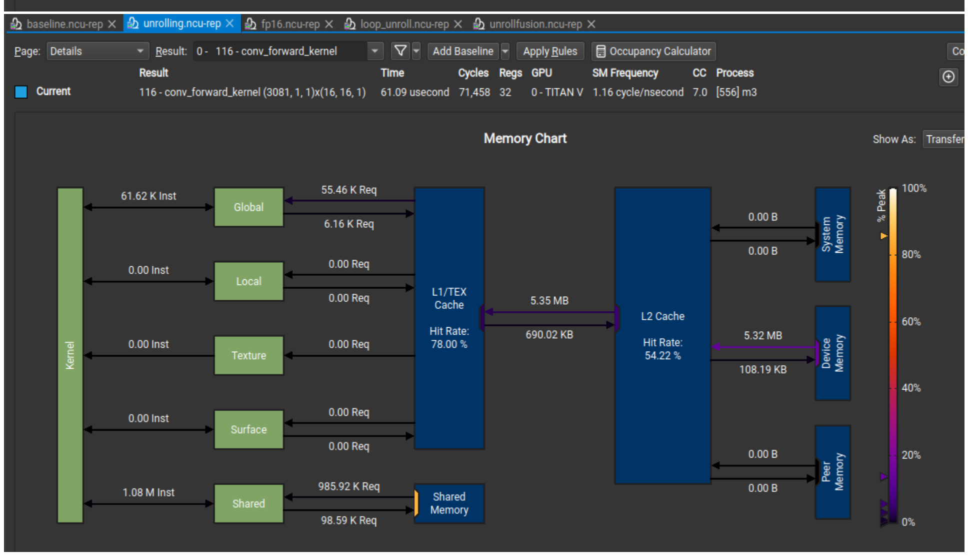


```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)    Total Time     Calls      Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
   64.1      411405901         20     20570295.1        26849    183150229  cudaMemcpy
   32.1      205896129         20     10294806.4         4018    201976175  cudaMalloc
    3.3       20904524         16      1306532.7          767     12872906  cudaDeviceSynchronize
    0.5        3079617         20       153980.9         3961       881426  cudaFree
    0.1         527186         10        52718.6        18745       288183  cudaLaunchKernel




Generating CUDA Kernel Statistics...

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

Time(%)    Total Time   Instances    Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
  100.0       21124970          6      3520828.3        17024     12871057  conv_forward_kernel
    0.0           2816          2         1408.0         1344         1472  do_not_remove_this_kernel
    0.0           2368          2         1184.0         1120         1248  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)    Total Time  Operations    Average       Minimum      Maximum    Name
-------  -------------  ---------  -------------  -----------  -----------  -------------------
   77.9      317591815         14     22685129.6         1152    183160913  [CUDA memcpy HtoD]
   22.1       90305107          6     15050851.2        23168     52478234  [CUDA memcpy DtoH]


CUDA Memory Operation Statistics (KiB)

          Total      Operations       Average          Minimum          Maximum    Name
---------------  ------------  ---------------  ---------------  ---------------  -------------------
      1911870.0            14         136562.2            0.004        1102500.0  [CUDA memcpy HtoD]
       156447.0             6          26074.0          148.535          90000.0  [CUDA memcpy DtoH]
```









*No. Because the convolution kernel, as expected, takes longer time due to the fact that we have to load the data into shared memory first, and the copying between shared memory and global memory takes time even longer than pure global memory access due to the fact that we have better memory coalescing in pure global memory access pattern because of my design. The exec time will also be larger because the input unroll makes the memory usage about K*K times larger than baseline. (As we can see from nsys, cudaMalloc and cudaMemcpy takes significantly longer time, conv_forward_kernel also takes longer time).*

*When we compare nsight compute GUI, we can see this optimization uses more memory bandwidth because we have much larger input because of input unroll (K*K larger, around 49 times)*

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

   The lecture and MP4. Copilot also helps.

2. **Optimization 2:** IMPL_UNROLLING_KERNEL_FUSION **Kernel fusion for unrolling and matrix-multiplication (requires previous optimization) (**2 points**)**

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

      I use kernel fusion for the original CPU unrolling to kernel input unrolling and I optimized the original matrix multiplication.

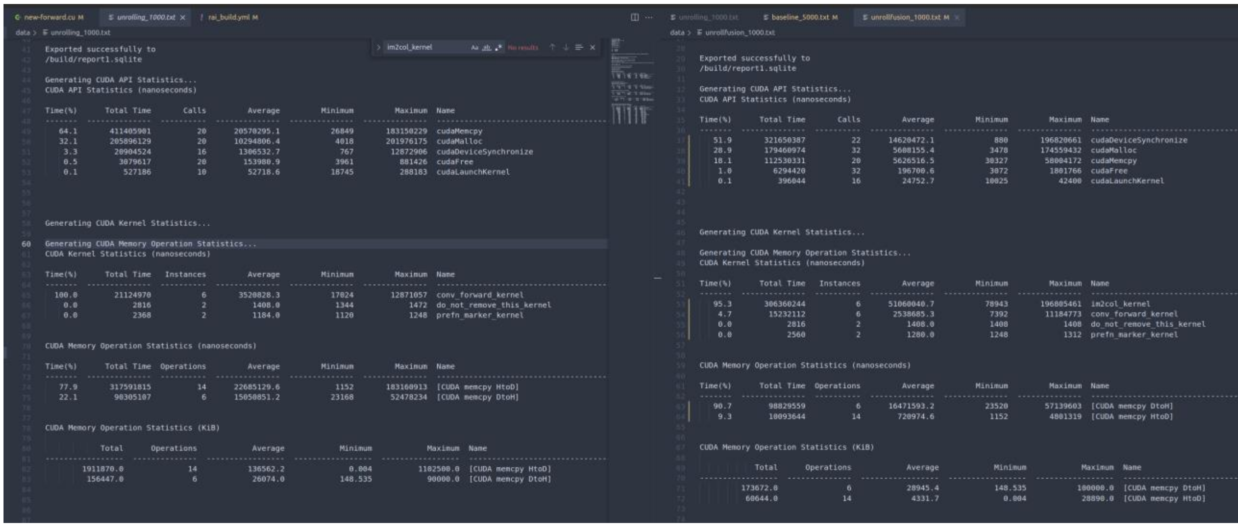
   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

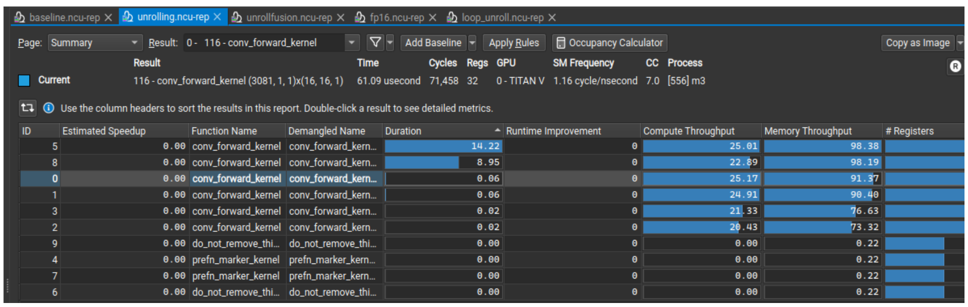
The optimization works by running input unrolling and finding a better way to do kernel multiplication. I think this will improve the op time as well as the exec time as the work was done in CPU in the previous optimization now it's done in GPU, and I tried the way with better memory coalescing. This optimization is built on previous optimization which gives a baseline of input unroll version implementation. In this optimization, the performance should be significantly improved.

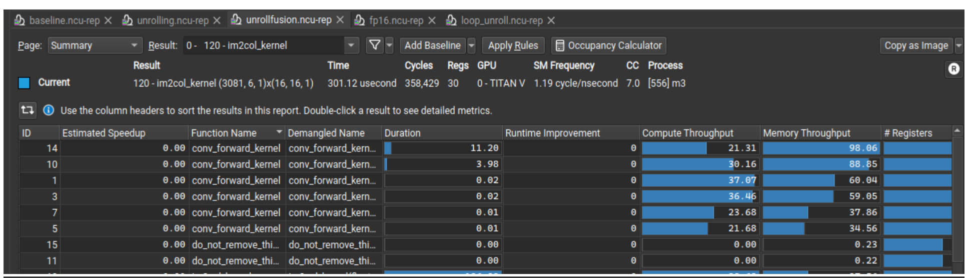
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

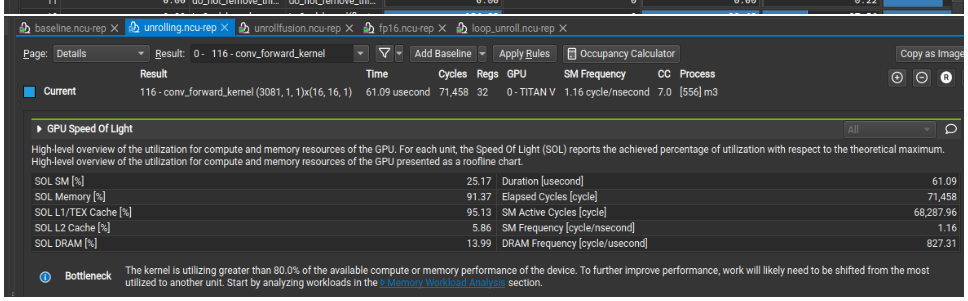
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.234637 ms*	*1.08158 ms*	*0m0.181s*	*0.86*
1000	*2.24452 ms*	*10.7243 ms*	*0m0.332s*	*0.886*
5000	*11.0265 ms*	*53.3184 ms*	*0m0.925s*	*0.871*

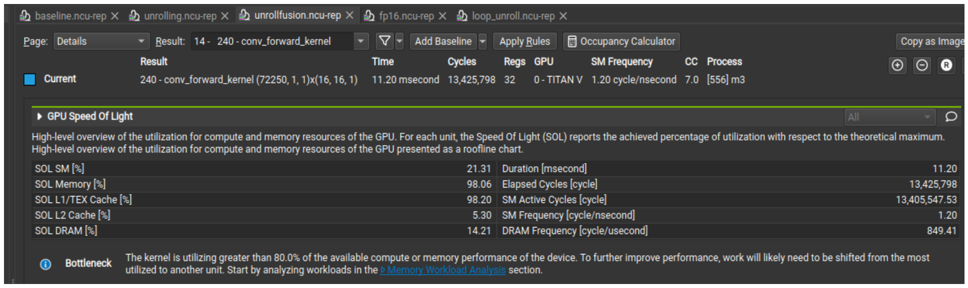
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).


new-forward.cu M
unrolling_1000.txt
rai_build.yml M
data > unrolling_1000.txt
im2col_kernel
No results
Exported successfully to
/build/report1.sqlite
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)
Time(%) Total Time Calls Average Minimum Maximum Name
64.1 411405981 20 20570295.1 26849 183150229 cudaMemcpy
32.1 205896129 20 10294806.4 4018 201976175 cudaMalloc
3.3 20904524 16 1306532.7 767 12872906 cudaDeviceSynchronize
0.5 3079617 20 153980.9 3961 881426 cudaFree
0.1 527186 10 52718.6 18745 288183 cudaLaunchKernel
Generating CUDA Kernel Statistics...
Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)
Time(%) Total Time Instances Average Minimum Maximum Name
100.0 21124970 6 3520828.3 17024 12871057 conv_forward_kernel
0.0 2816 2 1408.0 1344 1472 do_not_remove_this_kernel
0.0 2368 2 1184.0 1120 1248 prefn_marker_kernel
CUDA Memory Operation Statistics (nanoseconds)
Time(%) Total Time Operations Average Minimum Maximum Name
77.9 317591815 14 22685129.6 1152 183160913 [CUDA memcpy HtoD]
22.1 90305107 6 15050851.2 23168 52478234 [CUDA memcpy DtoH]
CUDA Memory Operation Statistics (KiB)
Total Operations Average Minimum Maximum Name
1911870.0 14 136562.2 0.004 1182500.0 [CUDA memcpy HtoD]
156447.0 6 26074.0 148.535 90000.0 [CUDA memcpy DtoH]
unrolling_1000.txt
baseline_5000.txt M
unrollfusion_1000.txt M
data > unrollfusion_1000.txt
Exported successfully to
/build/report1.sqlite
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)
Time(%) Total Time Calls Average Minimum Maximum Name
51.9 321650387 22 14620472.1 880 196820661 cudaDeviceSynchronize
28.9 179460974 32 5608155.4 3478 174559432 cudaMalloc
18.1 112530331 20 5626516.5 30327 58004172 cudaMemcpy
1.0 6294420 32 196700.6 3072 1801766 cudaFree
0.1 396044 16 24752.7 10025 42400 cudaLaunchKernel
Generating CUDA Kernel Statistics...
Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)
Time(%) Total Time Instances Average Minimum Maximum Name
95.3 306360244 6 51060040.7 78943 196805461 im2col_kernel
4.7 15232112 6 2538685.3 7392 11184773 conv_forward_kernel
0.0 2816 2 1408.0 1408 1408 do_not_remove_this_kernel
0.0 2560 2 1280.0 1248 1312 prefn_marker_kernel
CUDA Memory Operation Statistics (nanoseconds)
Time(%) Total Time Operations Average Minimum Maximum Name
90.7 98829559 6 16471593.2 23520 57139603 [CUDA memcpy DtoH]
9.3 10093644 14 720974.6 1152 4801319 [CUDA memcpy HtoD]
CUDA Memory Operation Statistics (KiB)
Total Operations Average Minimum Maximum Name
173672.0 6 28945.4 148.535 180000.0 [CUDA memcpy DtoH]
60644.0 14 4331.7 0.004 28890.0 [CUDA memcpy HtoD]


baseline.ncu-rep
unrolling.ncu-rep
unrollfusion.ncu-rep
fp16.ncu-rep
loop_unroll.ncu-rep
Page: Summary
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (3081, 1, 1)x(16, 16, 1) 61.09 usecond 71,458 32 0 - TITAN V 1.16 cycle/nsecond 7.0 [556] m3
Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.
ID Estimated Speedup Function Name Demangled Name Duration Runtime Improvement Compute Throughput Memory Throughput # Registers
5 0.00 conv_forward_kernel conv_forward_kern... 14.22 0 25.01 98.38
8 0.00 conv_forward_kernel conv_forward_kern... 8.95 0 22.89 98.19
0 0.00 conv_forward_kernel conv_forward_kern... 0.06 0 25.17 91.37
1 0.00 conv_forward_kernel conv_forward_kern... 0.06 0 24.91 90.40
3 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 21.33 76.63
2 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 20.43 73.32
9 0.00 do_not_remove_thi... do_not_remove_thi... 0.00 0 0.00 0.22
4 0.00 prefn_marker_kernel prefn_marker_kern... 0.00 0 0.00 0.22
7 0.00 prefn_marker_kernel prefn_marker_kern... 0.00 0 0.00 0.22
6 0.00 do_not_remove_thi... do_not_remove_thi... 0.00 0 0.00 0.22


baseline.ncu-rep
unrolling.ncu-rep
unrollfusion.ncu-rep
fp16.ncu-rep
loop_unroll.ncu-rep
Page: Summary
Result: 0 - 120 - im2col_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 120 - im2col_kernel (3081, 6, 1)x(16, 16, 1) 301.12 usecond 358,429 30 0 - TITAN V 1.19 cycle/nsecond 7.0 [556] m3
Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.
ID Estimated Speedup Function Name Demangled Name Duration Runtime Improvement Compute Throughput Memory Throughput # Registers
14 0.00 conv_forward_kernel conv_forward_kern... 11.20 0 21.31 98.06
10 0.00 conv_forward_kernel conv_forward_kern... 3.98 0 30.16 88.85
1 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 37.07 60.04
3 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 36.46 59.05
7 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 23.68 37.86
5 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 21.68 34.56
15 0.00 do_not_remove_thi... do_not_remove_thi... 0.00 0 0.00 0.23
11 0.00 do_not_remove_thi... do_not_remove_thi... 0.00 0 0.00 0.22


baseline.ncu-rep
unrolling.ncu-rep
unrollfusion.ncu-rep
fp16.ncu-rep
loop_unroll.ncu-rep
Page: Details
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (3081, 1, 1)x(16, 16, 1) 61.09 usecond 71,458 32 0 - TITAN V 1.16 cycle/nsecond 7.0 [556] m3
GPU Speed Of Light
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%] 25.17
SOL Memory [%] 91.37
SOL L1/TEX Cache [%] 95.13
SOL L2 Cache [%] 5.86
SOL DRAM [%] 13.99
Duration [usecond] 61.09
Elapsed Cycles [cycle] 71,458
SM Active Cycles [cycle] 68,287.96
SM Frequency [cycle/nsecond] 1.16
DRAM Frequency [cycle/usecond] 827.31
Bottleneck
The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.



Yes. It significantly improves both exec time and Op time. For exec time it's easy because we move CPU exec to GPU kernel exec. For Op time it's actually quite dependent on batch size. When the batch size isn't big enough, using shared memory would add extra overhead since the global memory isn't fully exploited or the over-utilized global memory bandwidth isn't great enough to degrade the performance of kernel fusion in this optimization, from the statistics, it could be seen that although the memory usage (global) in this optimization is greater than the previous optimization, the peformance is still better.

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

Asked chatgpt and copilot for some help.

3. Optimization 3: IMPL_LOOP_UNROLL **Tuning with restrict and loop unrolling (considered as one optimization only if you do both) (**3 points**)**

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

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

I added __restrict__ keyword to pointer and I also enumerate several cases of K (K=1,2,3,4,7) for unrolling purpose. The restrict pointer can help compiler improve the performance, and the unrolling can significantly improve the baseline approach by improving memory coalescing ability and helping compiler and reduce the use of extra registers which would cause many overheads.

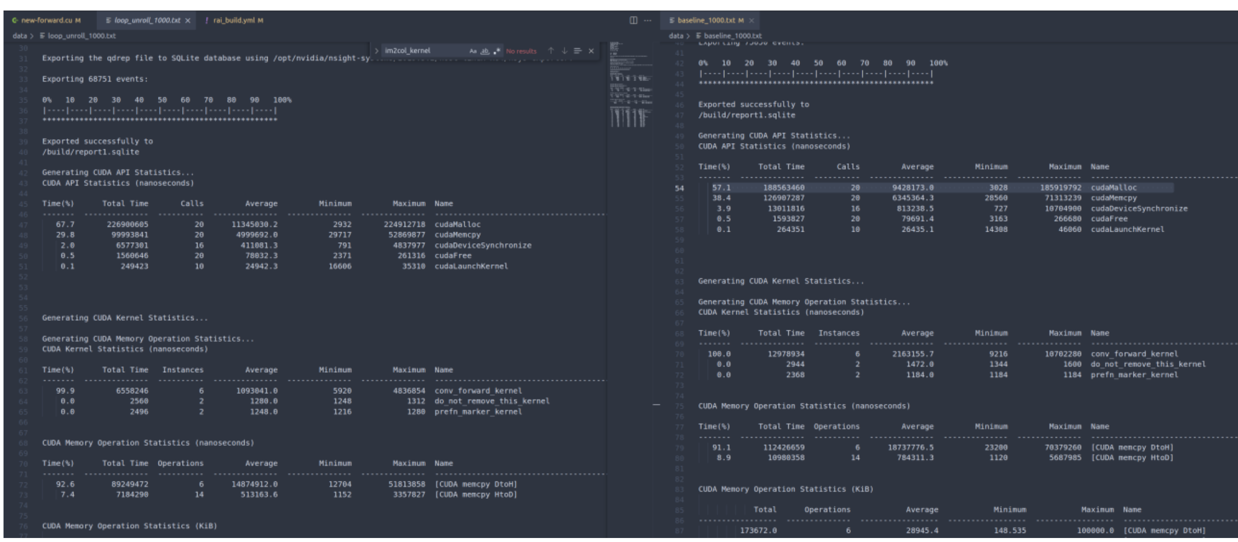
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

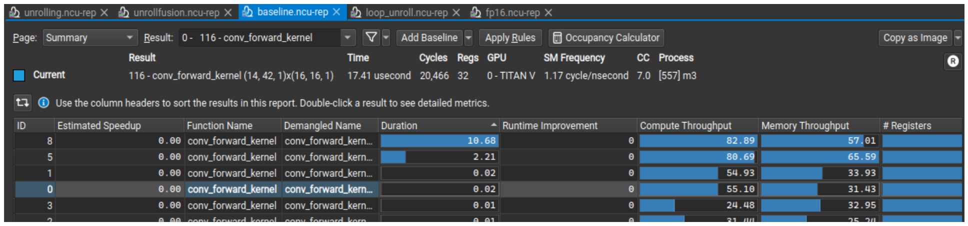
The restrict pointer can help compiler improve the performance by informing the compiler that this memory region is exclusive to this pointer and will not be referenced by other pointer in this kernel, allowing it to make optimization if available, and the unrolling can significantly improve the baseline approach by improving memory coalescing ability and helping compiler and reduce the use of extra registers which would cause many overheads. This would definitely improve the performance for forward convolution for previously stated reasons. This optimization is independent from previous 2 optimizations and should be compared with baseline implementation.

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

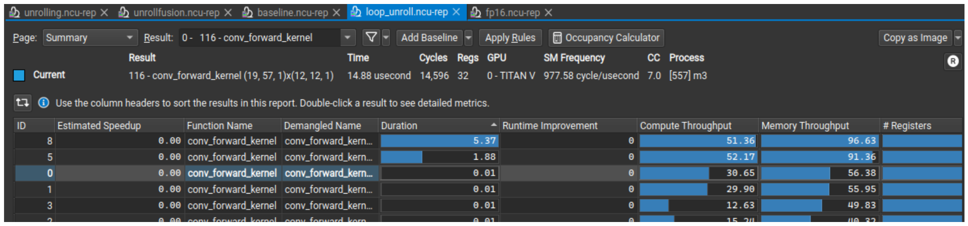
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.197214 ms*	*0.540495 ms*	*0m0.173s*	*0.86*
1000	*1.89108 ms*	*5.41518 ms*	*0m0.344s*	*0.886*
5000	*10.1522 ms*	*26.9867 ms*	*0m1.024s*	*0.871*

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

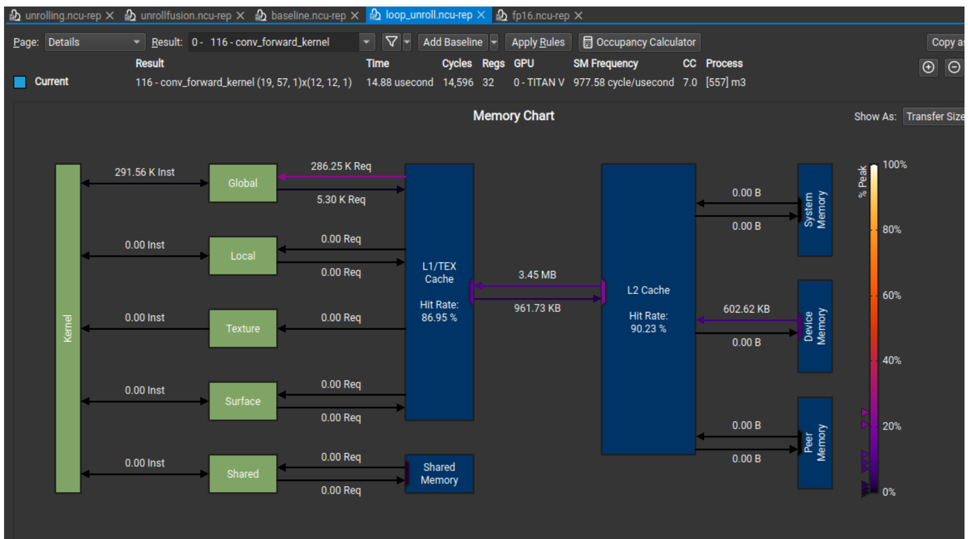




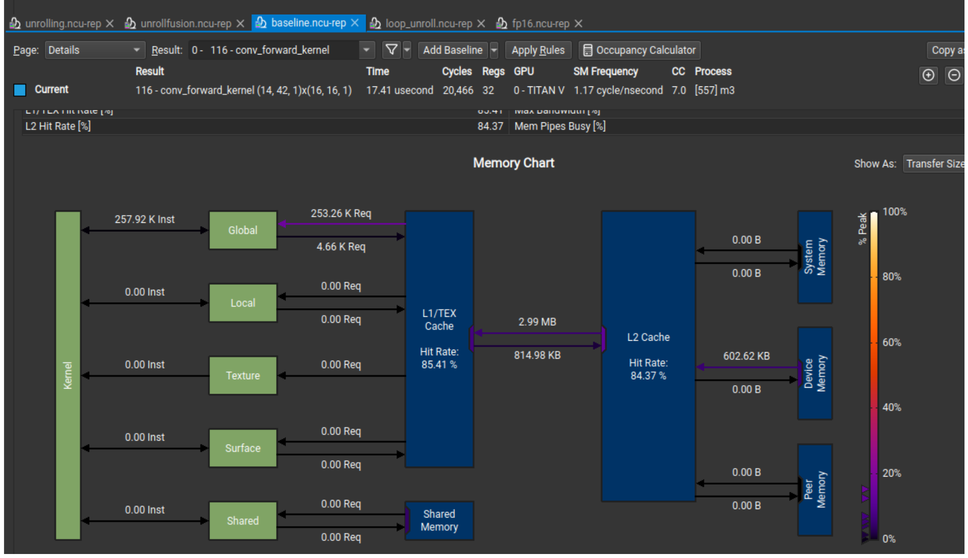
unrolling.ncu-rep
unrollfusion.ncu-rep
baseline.ncu-rep
loop_unroll.ncu-rep
fp16.ncu-rep
Page: Summary
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (14, 42, 1)x(16, 16, 1) 17.41 usecond 20,466 32 0 - TITAN V 1.17 cycle/nsecond 7.0 [557] m3
Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.
ID Estimated Speedup Function Name Demangled Name Duration Runtime Improvement Compute Throughput Memory Throughput # Registers
8 0.00 conv_forward_kernel conv_forward_kern... 10.68 0 82.89 57.01
5 0.00 conv_forward_kernel conv_forward_kern... 2.21 0 80.69 65.59
1 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 54.93 33.93
0 0.00 conv_forward_kernel conv_forward_kern... 0.02 0 55.10 31.43
3 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 24.48 32.95



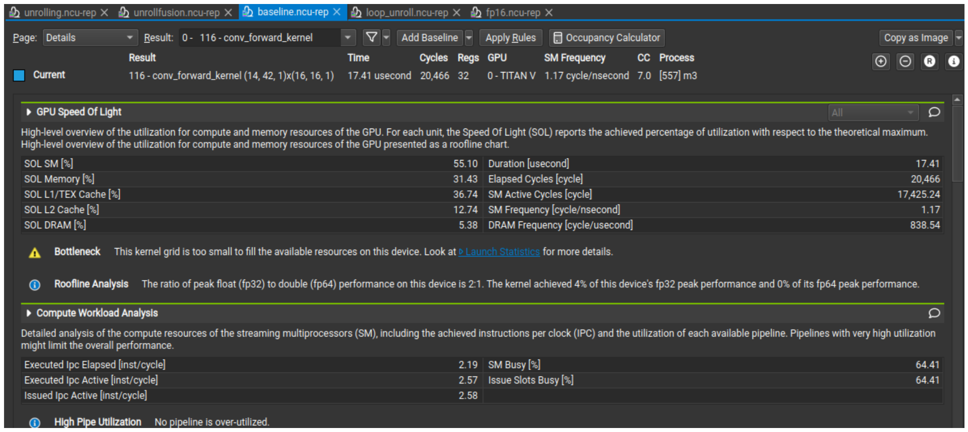
unrolling.ncu-rep
unrollfusion.ncu-rep
baseline.ncu-rep
loop_unroll.ncu-rep
fp16.ncu-rep
Page: Summary
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (19, 57, 1)x(12, 12, 1) 14.88 usecond 14,596 32 0 - TITAN V 977.58 cycle/usecond 7.0 [557] m3
Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.
ID Estimated Speedup Function Name Demangled Name Duration Runtime Improvement Compute Throughput Memory Throughput # Registers
8 0.00 conv_forward_kernel conv_forward_kern... 5.37 0 51.36 96.63
5 0.00 conv_forward_kernel conv_forward_kern... 1.88 0 52.17 91.36
0 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 30.65 56.38
1 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 29.90 55.95
3 0.00 conv_forward_kernel conv_forward_kern... 0.01 0 12.63 49.83

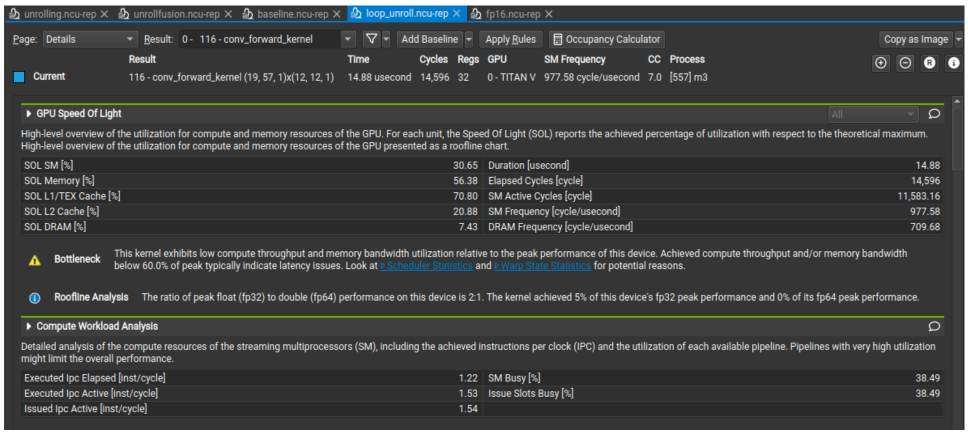


unrolling.ncu-rep
unrollfusion.ncu-rep
baseline.ncu-rep
loop_unroll.ncu-rep
fp16.ncu-rep
Page: Details
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (19, 57, 1)x(12, 12, 1) 14.88 usecond 14,596 32 0 - TITAN V 977.58 cycle/usecond 7.0 [557] m3
Memory Chart
Show As: Transfer Size
Kernel
291.56 K Inst
Global
286.25 K Req
5.30 K Req
0.00 Inst
Local
0.00 Req
0.00 Req
0.00 Inst
Texture
0.00 Req
0.00 Inst
Surface
0.00 Req
0.00 Req
0.00 Inst
Shared
0.00 Req
0.00 Req
L1/TEX Cache
Hit Rate: 86.95 %
Shared Memory
3.45 MB
961.73 KB
L2 Cache
Hit Rate: 90.23 %
0.00 B
0.00 B
System Memory
602.62 KB
0.00 B
Device Memory
0.00 B
0.00 B
Peer Memory
% Peak
100%
80%
60%
40%
20%
0%



unrolling.ncu-rep
unrollfusion.ncu-rep
baseline.ncu-rep
loop_unroll.ncu-rep
fp16.ncu-rep
Page: Details
Result: 0 - 116 - conv_forward_kernel
Add Baseline
Apply Rules
Occupancy Calculator
Result Time Cycles Regs GPU SM Frequency CC Process
Current 116 - conv_forward_kernel (14, 42, 1)x(16, 16, 1) 17.41 usecond 20,466 32 0 - TITAN V 1.17 cycle/nsecond 7.0 [557] m3
L2 Hit Rate [%] 84.37 Mem Pipes Busy [%]
Memory Chart
Show As: Transfer Size
Kernel
257.92 K Inst
Global
253.26 K Req
4.66 K Req
0.00 Inst
Local
0.00 Req
0.00 Req
0.00 Inst
Texture
0.00 Req
0.00 Inst
Surface
0.00 Req
0.00 Req
0.00 Inst
Shared
0.00 Req
0.00 Req
L1/TEX Cache
Hit Rate: 85.41 %
Shared Memory
2.99 MB
814.98 KB
L2 Cache
Hit Rate: 84.37 %
0.00 B
0.00 B
System Memory
602.62 KB
0.00 B
Device Memory
0.00 B
0.00 B
Peer Memory
% Peak
100%
80%
60%
40%
20%
0%





The implementation is very successful in terms of performance (Op time) because the memory is better coalesced which can be shown that SOL memory cycles is less in current optimization. However, the total data transferred from memory should (trivially) be unchanged as shown.

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

4. **Optimization 4:** IMPL_FP16 **FP16 arithmetic. (note this can modify model accuracy slightly) (**4 points**)** ***(Delete this section blank if you did not implement this many optimizations.)***

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

FP16 arithmetic with restrict and loop unrolling because this set of optimizations should be most promising since all the operations can operate on 16bit width operands which will significantly reduce overhead. Although it's note-worthy that the conversion of input and mask and output can introduce overhead.

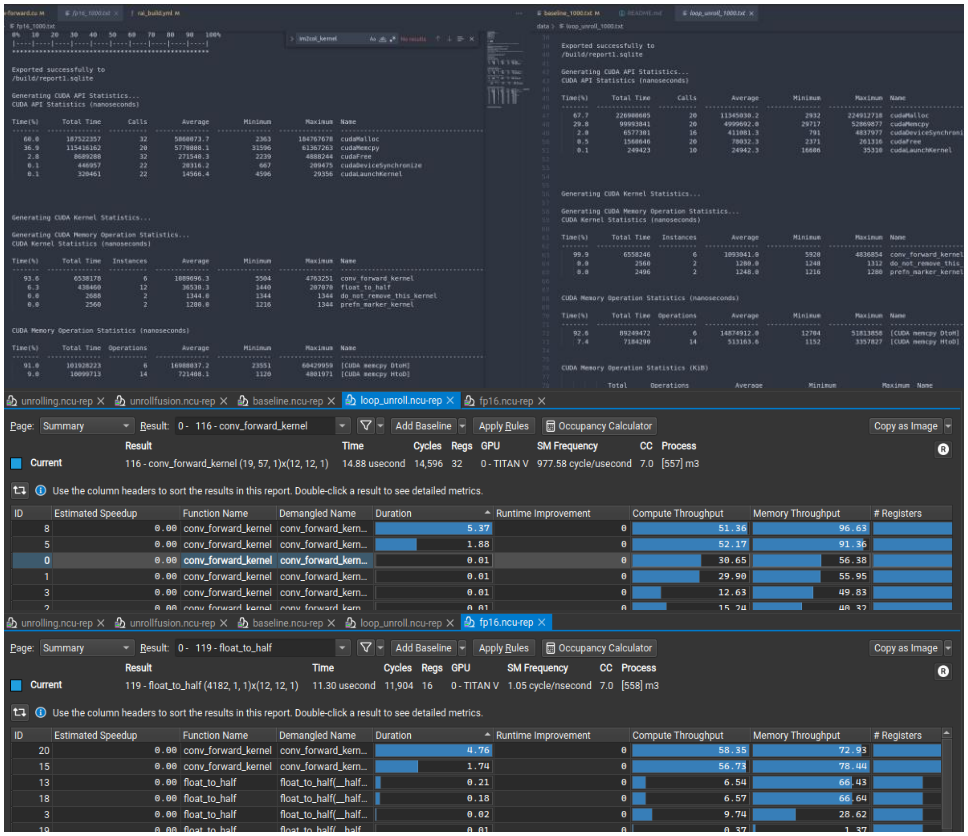
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

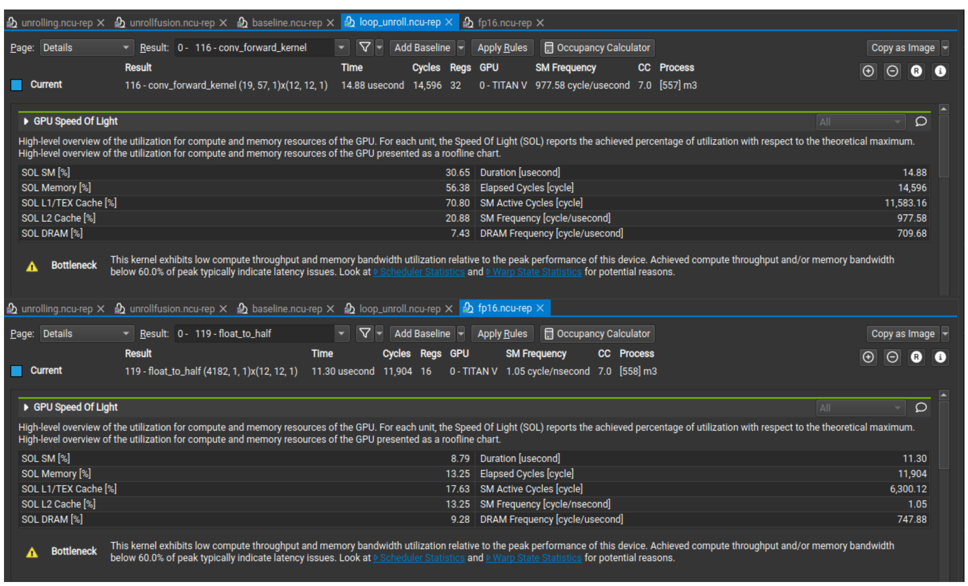
It works by transforming input matrix and mask matrix to FP16 accuracy. I think it will reduce the overhead of arithmetic calculation and reduce the memory throughput. This optimization is built from the loop unrolling optimization and is expected to reach the best performance.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.238143 ms*	*0.519883 ms*	*0m0.228s*	*0.86*
1000	*2.22948 ms*	*5.2218 ms*	*0m0.316s*	*0.887*
5000	*10.6534 ms*	*25.0527 ms*	*0m0.904s*	*0.8712*

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





From result from NSYS and nsight compute we can see FP16 reduce the overhead of arithmetic operations as well as memory throughput as expected. This becomes my best implementation and is submitted with this version.

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