Name: Benoit Ortalo-Magne, Leon Medvinsky, Steven Lee
NetId: beo2, leonkm2, solee2
Team name: ChampaignWithoutTheCham
School: Illinois

Milestone 2

```
* Running /bin/bash -c "./m2 10000"
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-CPU==
Op Time: 89215.8 ms
Conv-CPU==
Op Time: 263190 ms
Test Accuracy: 0.8714
```

Test Accuracy: .8714
Op Times: 83972.5 ms and 243484 ms
Total Time: 327456.5 ms

Milestone 3

Rai running your GPU implementation of convolution:

```
* Running bin/bash -c "./m3 10000"
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Conv-GPU==

Test Accuracy: 0.8714
--------------------------------
-            TIMINGS
--------------------------------
Layer 1 GPUTime: 38.464247 ms
Layer 1 OpTime: 38.473143 ms
Layer 1 LayerTime: 631.439901 ms
Layer 2 GPUTime: 151.761218 ms
Layer 2 OpTime: 151.77021 ms
Layer 2 LayerTime: 592.337754 ms
```

Nsys profiling GPU execution:

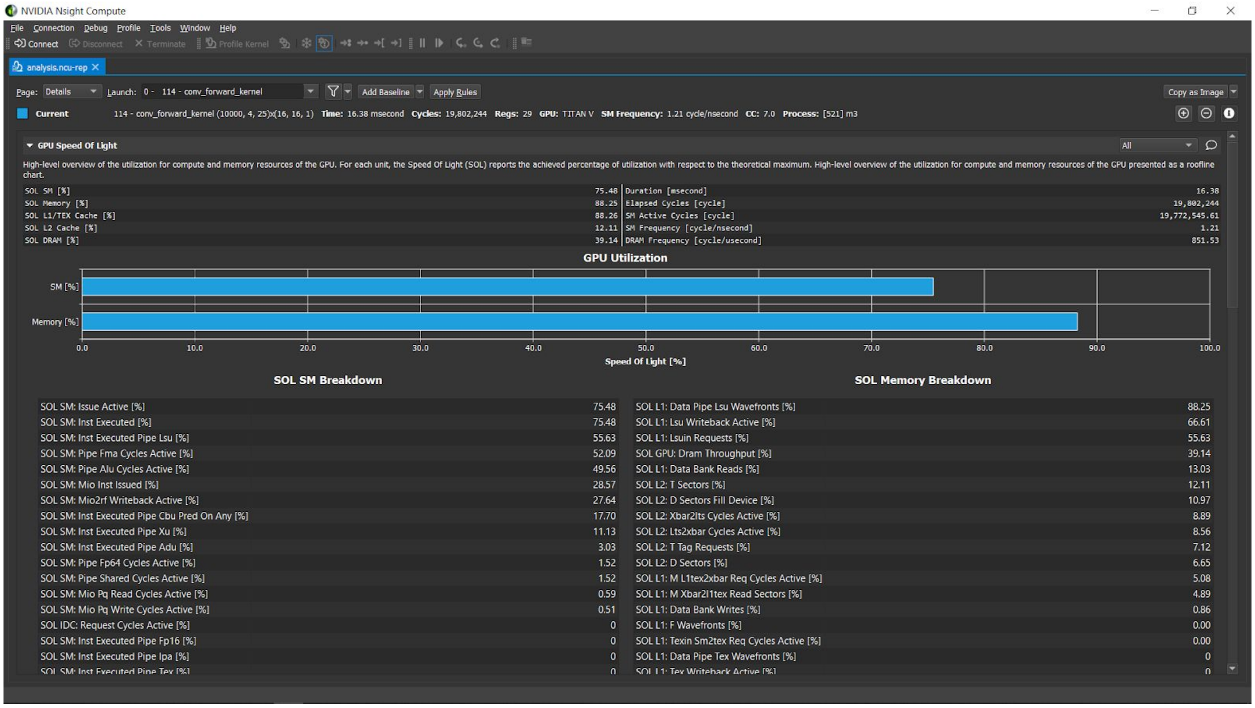
```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   85.9      1118091706           6     186348617.7           85329       535389563  cudaMemcpy
   13.7       178500915           6      29750152.5           64024       176991203  cudaMalloc
    0.3         4431407           6        738567.8           58046         2849717  cudaFree
    0.0          262923           2        131461.5           25790          237133  cudaLaunchKernel
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0       138749857           2      69374928.5        16497215       122252642  conv_forward_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.6       895095099           2     447547549.5       376998837       518096262  [CUDA memcpy DtoH]
    7.4        71795884           4      17948971.0            1248        38466642  [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               4           134729.0              0.766           288906.0  [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     94840706872         963      98484638.5           51075       100207899  sem_timedwait
   33.3     94788430779         961      98635203.7           57366       100773798  poll
   22.1     62788851699           2   31394425849.5     22305165637     40483686062  pthread_cond_wait
   11.2     32009136178          64     500142752.8       500084730       500210532  pthread_cond_timedwait
    0.0       101001684         764        132201.2            1063        15635483  ioctl
    0.0        19053488        9072          2100.3            1220           18841  read
    0.0         2712393          97         27962.8            1043          970934  mmap
    0.0          627899          97          6473.2            1716           21485  open64
    0.0          625833           2        312916.5           40823          585010  pthread_mutex_lock
    0.0          220436           5         44087.2           31233           53756  pthread_create
```

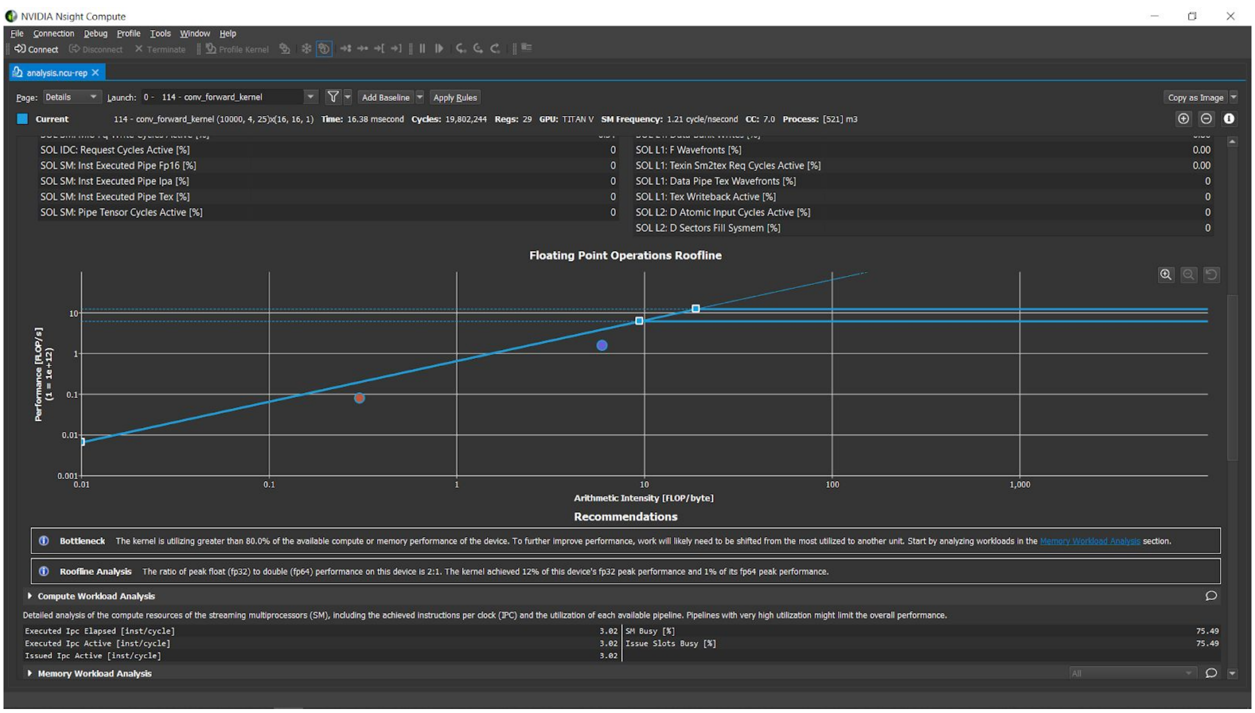
Kernels that consume more than 90% of program time: conv_forward_kernel

CUDA API calls that consume more than 90% of program time: cudaMemcpy, cudaMalloc, cudaLaunchKernel, cudaFree

Difference between kernels and API calls: API calls are used to set up memory and variables for kernels to run on.

Screenshot of GPU SOL utilization:


NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel
analysis.ncu-rep
Page: Details Launch: 0 - 114 - conv_forward_kernel Add Baseline Apply Rules Copy as Image
Current 114 - conv_forward_kernel (10000, 4, 25)x(16, 16, 1) Time: 16.38 msecond Cycles: 19,802,244 Regs: 29 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [521] 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 [%] 75.48
SOL Memory [%] 88.25
SOL L1/TEX Cache [%] 88.26
SOL L2 Cache [%] 12.11
SOL DRAM [%] 39.14
Duration [msecond] 16.38
Elapsed Cycles [cycle] 19,802,244
SM Active Cycles [cycle] 19,772,545.61
SM Frequency [cycle/nsecond] 1.21
DRAM Frequency [cycle/usecond] 851.53
GPU Utilization
SM [%]
Memory [%]
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0
Speed Of Light [%]
SOL SM Breakdown
SOL SM: Issue Active [%] 75.48
SOL SM: Inst Executed [%] 75.48
SOL SM: Inst Executed Pipe Lsu [%] 55.63
SOL SM: Pipe Fma Cycles Active [%] 52.09
SOL SM: Pipe Alu Cycles Active [%] 49.56
SOL SM: Mio Inst Issued [%] 28.57
SOL SM: Mio2rf Writeback Active [%] 27.64
SOL SM: Inst Executed Pipe Cbu Pred On Any [%] 17.70
SOL SM: Inst Executed Pipe Xu [%] 11.13
SOL SM: Inst Executed Pipe Adu [%] 3.03
SOL SM: Pipe Fp64 Cycles Active [%] 1.52
SOL SM: Pipe Shared Cycles Active [%] 1.52
SOL SM: Mio Pq Read Cycles Active [%] 0.59
SOL SM: Mio Pq Write Cycles Active [%] 0.51
SOL IDC: Request Cycles Active [%] 0
SOL SM: Inst Executed Pipe Fp16 [%] 0
SOL SM: Inst Executed Pipe Ipa [%] 0
SOL Memory Breakdown
SOL L1: Data Pipe Lsu Wavefronts [%] 88.25
SOL L1: Lsu Writeback Active [%] 66.61
SOL L1: Lsuin Requests [%] 55.63
SOL GPU: Dram Throughput [%] 39.14
SOL L1: Data Bank Reads [%] 13.03
SOL L2: T Sectors [%] 12.11
SOL L2: D Sectors Fill Device [%] 10.97
SOL L2: Xbar2lts Cycles Active [%] 8.89
SOL L2: Lts2xbar Cycles Active [%] 8.56
SOL L2: T Tag Requests [%] 7.12
SOL L2: D Sectors [%] 6.65
SOL L1: M L1tex2xbar Req Cycles Active [%] 5.08
SOL L1: M Xbar2l1tex Read Sectors [%] 4.89
SOL L1: Data Bank Writes [%] 0.86
SOL L1: F Wavefronts [%] 0.00
SOL L1: Texin Sm2tex Req Cycles Active [%] 0.00
SOL L1: Data Pipe Tex Wavefronts [%] 0


NVIDIA Nsight Compute
File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel
analysis.ncu-rep
Page: Details Launch: 0 - 114 - conv_forward_kernel Add Baseline Apply Rules Copy as Image
Current 114 - conv_forward_kernel (10000, 4, 25)x(16, 16, 1) Time: 16.38 msecond Cycles: 19,802,244 Regs: 29 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [521] m3
SOL IDC: Request Cycles Active [%] 0
SOL SM: Inst Executed Pipe Fp16 [%] 0
SOL SM: Inst Executed Pipe Ipa [%] 0
SOL SM: Inst Executed Pipe Tex [%] 0
SOL SM: Pipe Tensor Cycles Active [%] 0
SOL L1: F Wavefronts [%] 0.00
SOL L1: Texin Sm2tex Req Cycles Active [%] 0.00
SOL L1: Data Pipe Tex Wavefronts [%] 0
SOL L1: Tex Writeback Active [%] 0
SOL L2: D Atomic Input Cycles Active [%] 0
SOL L2: D Sectors Fill Sysmem [%] 0
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
10 1 0.1 0.01 0.001
0.01 0.1 1 10 100 1,000
Arithmetic Intensity [FLOP/byte]
Recommendations
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.
Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 12% of this device's fp32 peak performance and 1% of its fp64 peak performance.
Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle] 3.02
Executed Ipc Active [inst/cycle] 3.02
Issued Ipc Active [inst/cycle] 3.02
SM Busy [%] 75.49
Issue Slots Busy [%] 75.49
Memory Workload Analysis

Milestone 4

1. Shared memory

We noticed that the baseline kernel has high memory bandwidth utilization compared to SM utilization, suggesting the kernel was memory-bound. Since convolution (especially batched in 3d) involves a lot of memory reuse between threads, using shared memory to reduce the number of global memory reads was an obvious way to improve the kernel's memory bandwidth requirements.

Effect

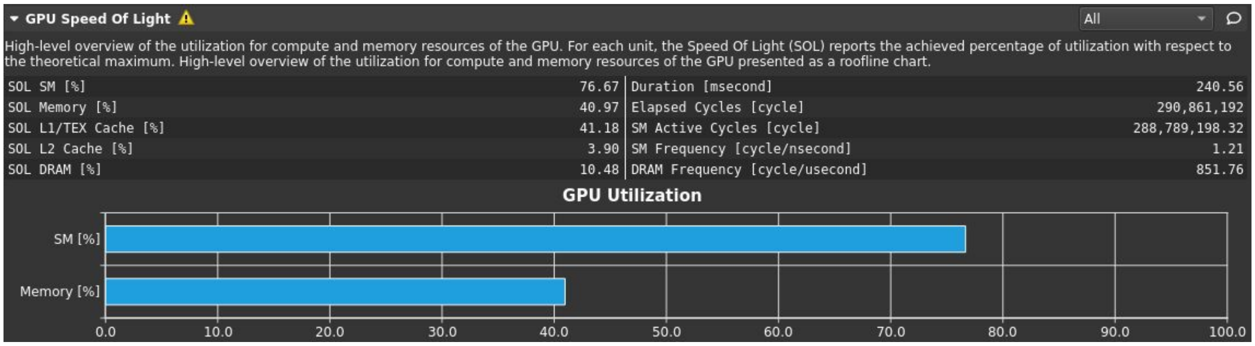
Overall timing information is as follows:

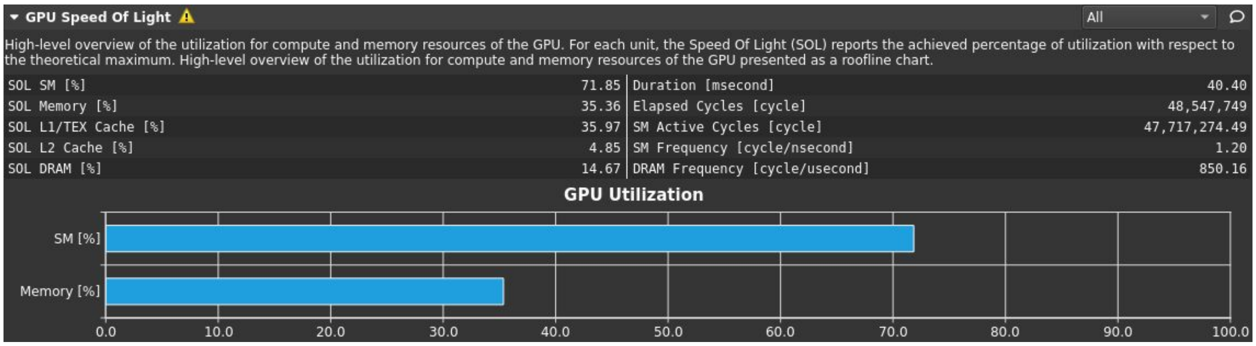
```
Layer 1 GPUTime: 40.26357 ms
Layer 1 OpTime: 40.302002 ms
Layer 1 LayerTime: 638.512441 ms
Layer 2 GPUTime: 226.928337 ms
Layer 2 OpTime: 226.961872 ms
Layer 2 LayerTime: 681.999028 ms
```

Compared to the baseline, performance is considerably worse. Nsys does not shed much light on this, mainly showing the increased execution time.

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   81.4      1317579041           8     164697380.1           15431       628180643  cudaMemcpy
   17.4       282160770           8      35270096.2           74170       272446462  cudaMalloc
    1.0        15821774           6       2636962.3           18114        15696648  cudaLaunchKernel
    0.2         2767247           8        345905.9           72194          975341  cudaFree
    0.0           22261           4          5565.2            2393           12987  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0       277074773           2     138537386.5        40595080       236479693  conv_forward_kernel
    0.0            2688           2          1344.0            1312            1376  do_not_remove_this_kernel
    0.0            2592           2          1296.0            1248            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
   92.8       960835387           2     480417693.5       390890544       569944843  [CUDA memcpy DtoH]
    7.2        74126576           6      12354429.3            1216        38931280  [CUDA memcpy HtoD]
```

The output of Nsight, on the other hand, shows that memory utilization is now significantly lower than SM utilization, suggesting that the computation has become compute-bound rather than memory-bound:





We interpreted this as the memory bandwidth requirements of the kernel having been reduced, but the additional overhead, from computation, use of shared memory, and synchronization, being enough to decrease performance. Despite this, the results of profiling indicate that using shared memory will be useful in exposing opportunities to optimize computational resource use.

References

We followed the textbook's strategy for using shared memory in batched convolution.

2. Constant Memory

Since the kernel values are never modified, we can leverage constant memory to read values faster and further optimize the execution time.

How you identified the optimization opportunity: kernel memory is made up of constant values so they can be stored in constant memory to reduce load times while running the kernel

Why the approach is fruitful: It allows for faster load times when performing calculations using kernel memory

The effect: Overall, the time taken is decreased. This implementation improves timing across layers, making it a clear improvement.

With constant memory

```
Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 14.879225 ms
Layer 1 OpTime: 14.902041 ms
Layer 1 LayerTime: 605.238437 ms
Layer 2 GPUTime: 54.966667 ms
Layer 2 OpTime: 55.004011 ms
Layer 2 LayerTime: 481.50531 ms
```

Base Implementation

```
Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 16.54464 ms
Layer 1 OpTime: 16.574783 ms
Layer 1 LayerTime: 650.947637 ms
Layer 2 GPUTime: 58.609756 ms
Layer 2 OpTime: 58.639516 ms
Layer 2 LayerTime: 521.298612 ms
```

References: Lecture slides

Analysis using nsys:

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   85.5      1284671642           6     214111940.3           16040       625229124  cudaMemcpy
   13.3       199828379           8      24978547.4           80100       196121320  cudaMalloc
    1.0        14442027           6       2407004.5           16596        14329357  cudaLaunchKernel
    0.2         2824313           8        353039.1           69063          871088  cudaFree
    0.0          168240           2         84120.0           83535           84705  cudaMemcpyToSymbol
    0.0           20408           4          5102.0            2418            7891  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
  100.0       270338390           2     135169195.0        43672815       226665575  conv_forward_kernel
    0.0            2752           2          1376.0            1312            1440  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1312            1344  prefn_marker_kernel
```

This nsys output shows runtimes for the kernel with both constant memory and shared memory optimization. The first data set shows that even though we add a cudaMemcpyToSymbol call, the time it adds is worth the tradeoff as it significantly reduces the time other calls take. The second data set shows the total time our conv_forward_kernel takes which, again, is faster than the previous runs. This leads me to the conclusion that constant memory is worth implementing and speeds up the program's runtime.

3. Tuning with restrict and loop unrolling (considered as one optimization only if you do both)

We noticed that our for loops could be unrolled to obtain better performance results. This approach seemed fruitful because we could essentially half the number of iterations of the for loop and each iteration would do double the work. After implementing loop unrolling we found that the results were indeed fruitful:

After loop-unrolling:

Before loop-unrolling (w/ shared & constant mem):

```
Test Accuracy: 0.8714
--------------------------------
-             TIMINGS
--------------------------------
Layer 1 GPUTime: 38.660198 ms
Layer 1 OpTime: 38.686886 ms
Layer 1 LayerTime: 685.301769 ms
Layer 2 GPUTime: 202.451139 ms
Layer 2 OpTime: 202.481731 ms
Layer 2 LayerTime: 648.082145 ms
```

```
Test Accuracy: 0.8714
--------------------------------
-             TIMINGS
--------------------------------
Layer 1 GPUTime: 43.350917 ms
Layer 1 OpTime: 43.374981 ms
Layer 1 LayerTime: 668.012451 ms
Layer 2 GPUTime: 223.536424 ms
Layer 2 OpTime: 223.568904 ms
Layer 2 LayerTime: 691.185513 ms
```

We utilized the Nvidia documentation on unrolling loops [here](https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf).
(https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf)

Additionally, we used __restrict__ to promise to the compiler that no pointer aliasing occurs. We pair this with using the keyword "const" as well to ensure that the input pointers are read only. After pairing this with loop-unrolling, we got these results which show further improvement than just loop-unrolling alone:

```
Test Accuracy: 0.8714
--------------------------------
-             TIMINGS
--------------------------------
Layer 1 GPUTime: 36.651829 ms
Layer 1 OpTime: 36.674325 ms
Layer 1 LayerTime: 628.448989 ms
Layer 2 GPUTime: 203.304852 ms
Layer 2 OpTime: 203.331796 ms
Layer 2 LayerTime: 642.932878 ms
```

We utilized the Nvidia documentation [here](https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/) and [this Piazza post](https://piazza.com/class/kdqhmvil9xq67p?cid=491).
(https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/)
(https://piazza.com/class/kdqhmvil9xq67p?cid=491)

According to the CUDA kernel statistics from nsys, our total time, average time, minimum time, and maximum time in conv_forward_kernel have all gone down as well.

Here are the nsys time statistics before restrict and loop-unrolling:

```
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0       274269946           2     137134973.0        43529320       230740626  conv_forward_kernel
    0.0            2720           2          1360.0            1312            1408  prefn_marker_kernel
    0.0            2624           2          1312.0            1248            1376  do_not_remove_this_kernel
```

Here are the nsys time statistics after restrict and loop-unrolling:

```
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------
  100.0       230854050           2     115427025.0        38140081       192713969  conv_forward_kernel
    0.0            2464           2          1232.0            1184            1280  prefn_marker_kernel
    0.0            2464           2          1232.0            1216            1248  do_not_remove_this_kernel
```

Additional Optimizations

Based on our analyses, we built our combined kernel with the following configuration:

- No shared memory
- An additional guard to reduce computation in some cases
- Reduced branching when unrolling

Organization (PM4)

We had each team member implement one optimization, measuring and writing up the analysis for the results on a shared Google doc. The parts of the report for previous PMs was taken from Benoit and Steven’s report.

The optimizations were assigned as follows:

- Benoit - Constant memory
- Steven - loop unrolling + restrict
- Leon - Shared memory (convolution)

Final Submission

1. FP16

FP16 can be used to make computations faster by allowing the processor to use less memory. Using the __half2 data type, we can perform two operations in a single register.

How you identified the optimization opportunity: The GPU does many arithmetic operations so speeding those up could result in a significant faster runtime.

Why the approach is fruitful: It allows for faster arithmetic computation times in the GPU.

The effect: Overall, the total time taken is very similar and OpTime is shown to be consistently a bit slower. I believe this is because the time taken to transform the values into __half2's is similar to the time gained by having more efficient arithmetic. (This run was made on top of previous constant memory optimization and has slight variations due to shared usage of rai GPU)

With FP16

```
Test Accuracy: 0.8716
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 20.221661 ms
Layer 1 OpTime: 20.245853 ms
Layer 1 LayerTime: 619.425948 ms
Layer 2 GPUTime: 72.630452 ms
Layer 2 OpTime: 72.661076 ms
Layer 2 LayerTime: 500.675067 ms
```

Base implementation

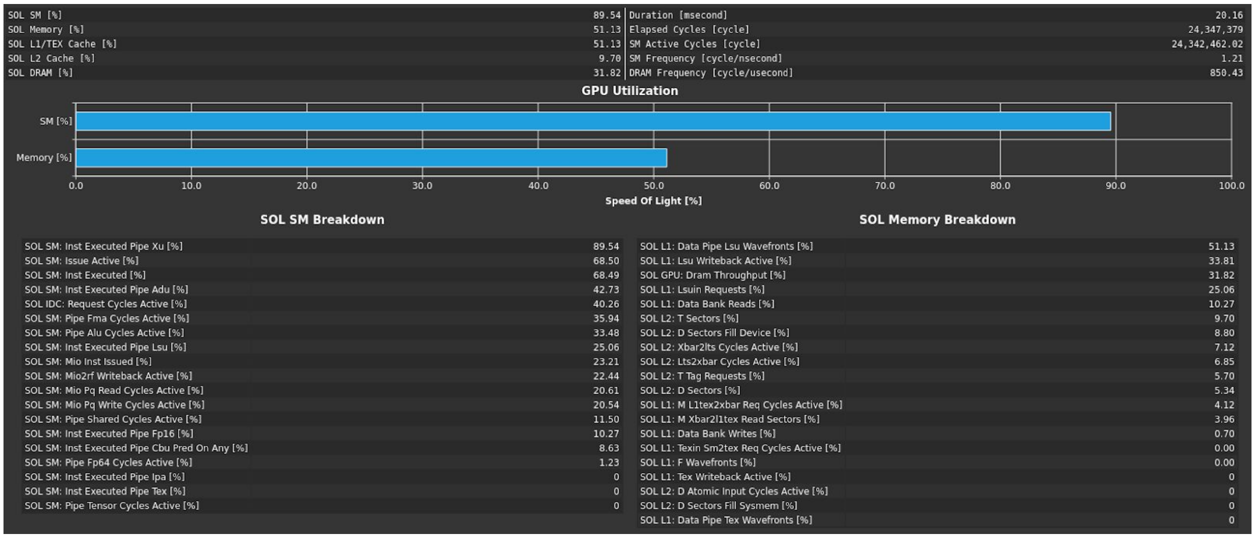
```
Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 17.439955 ms
Layer 1 OpTime: 17.445587 ms
Layer 1 LayerTime: 657.218768 ms
Layer 2 GPUTime: 57.648491 ms
Layer 2 OpTime: 57.681419 ms
Layer 2 LayerTime: 531.559798 ms
```

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

Nsys profiling:

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   78.2      1154380961           6     192396826.8           12190       595379172  cudaMemcpy
   19.9       293716529           8      36714566.1           65110       283262850  cudaMalloc
    1.6        22973183           6       3828863.8           20383        22843141  cudaLaunchKernel
    0.3         4302010           8        537751.3           62683         1901363  cudaFree
    0.0          170611           2         85305.5           79983           90628  cudaMemcpyToSymbol
    0.0           20382           4          5095.5            2346           10860  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
  100.0        99316539           2      49658269.5        20211741        79104798  conv_forward_kernel
    0.0            2848           2          1424.0            1344            1504  do_not_remove_this_kernel
    0.0            2592           2          1296.0            1248            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   93.2       975187745           2     487593872.5       400908765       574278980  [CUDA memcpy DtoH]
    6.8        71072613           6      11845435.5            1440        37264362  [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]
Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------------
   33.4     96940081579         983      98616563.2           26805       102484943  sem_timedwait
   33.1     96090883747         974      98655938.1           37483       100884043  poll
   22.1     64025321315           2   32012660657.5     23809445709     40215875606  pthread_cond_wait
   11.0     32008771817          64     500137059.6       500056944       500172939  pthread_cond_timedwait
    0.3       725794388         101       7186083.0            3418       724679098  open64
    0.1       216241321         866        249701.3            1151       103824375  ioctl
    0.0        20531431        9072          2263.2            1123           18684  read
    0.0         3653754          98         37283.2            1151         1686661  mmap
    0.0          267227           5         53445.4           36316           60382  pthread_create
    0.0          101327          18          5629.3            1262           15990  munmap
    0.0           99071          24          4128.0            1089           14355  fopen
    0.0           94821          15          6321.4            2539           14748  write
    0.0           78130           3         26043.3            3444           43684  fopen64
    0.0           76458           3         25486.0           11230           50078  fgets
```

Nsight profiling:



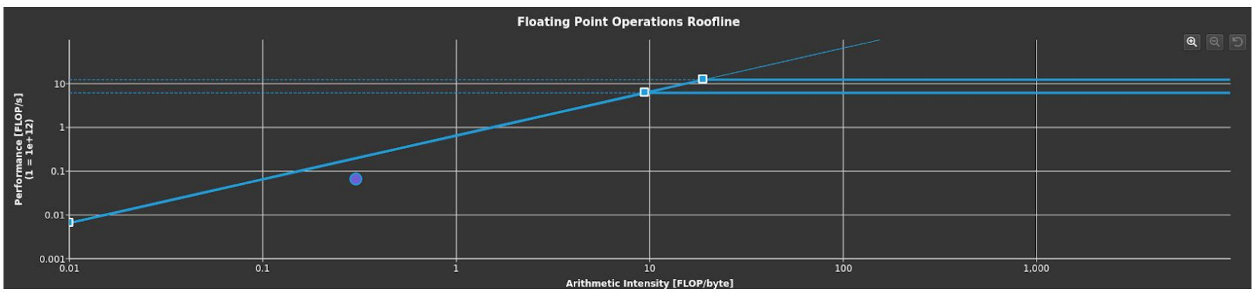
SOL SM [%]	89.54	Duration [msecond]	20.16
SOL Memory [%]	51.13	Elapsed Cycles [cycle]	24,347,379
SOL L1/TEX Cache [%]	51.13	SM Active Cycles [cycle]	24,342,462.02
SOL L2 Cache [%]	9.70	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	31.82	DRAM Frequency [cycle/usecond]	850.43

SOL SM Breakdown

SOL SM: Inst Executed Pipe Xu [%]	89.54
SOL SM: Issue Active [%]	68.50
SOL SM: Inst Executed [%]	68.49
SOL SM: Inst Executed Pipe Adu [%]	42.73
SOL IDC: Request Cycles Active [%]	40.26
SOL SM: Pipe Fma Cycles Active [%]	35.94
SOL SM: Pipe Alu Cycles Active [%]	33.48
SOL SM: Inst Executed Pipe Lsu [%]	25.06
SOL SM: Mio Inst Issued [%]	23.21
SOL SM: Mio2rf Writeback Active [%]	22.44
SOL SM: Mio Pq Read Cycles Active [%]	20.61
SOL SM: Mio Pq Write Cycles Active [%]	20.54
SOL SM: Pipe Shared Cycles Active [%]	11.50
SOL SM: Inst Executed Pipe Fp16 [%]	10.27
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	8.63
SOL SM: Pipe Fp64 Cycles Active [%]	1.23
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

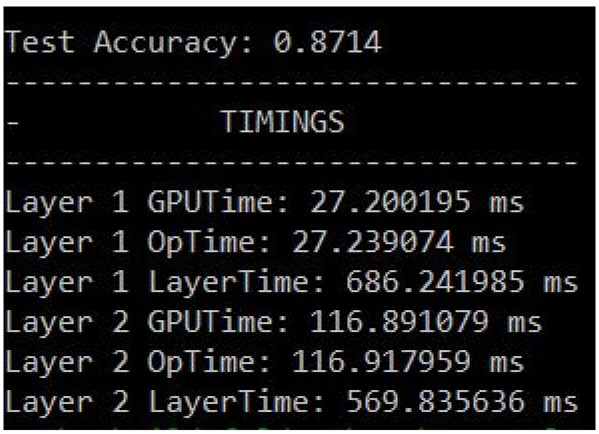
SOL L1: Data Pipe Lsu Wavefronts [%]	51.13
SOL L1: Lsu Writeback Active [%]	33.81
SOL GPU: Dram Throughput [%]	31.82
SOL L1: Lsuin Requests [%]	25.06
SOL L1: Data Bank Reads [%]	10.27
SOL L2: T Sectors [%]	9.70
SOL L2: D Sectors Fill Device [%]	8.80
SOL L2: Xbar2lts Cycles Active [%]	7.12
SOL L2: Lts2xbar Cycles Active [%]	6.85
SOL L2: T Tag Requests [%]	5.70
SOL L2: D Sectors [%]	5.34
SOL L1: M L1tex2xbar Req Cycles Active [%]	4.12
SOL L1: M Xbar2l1tex Read Sectors [%]	3.96
SOL L1: Data Bank Writes [%]	0.70
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L1: F Wavefronts [%]	0.00
SOL L1: Tex Writeback Active [%]	0
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0



2. Sweeping various parameters to find best values (block sizes, amount of thread coarsening)

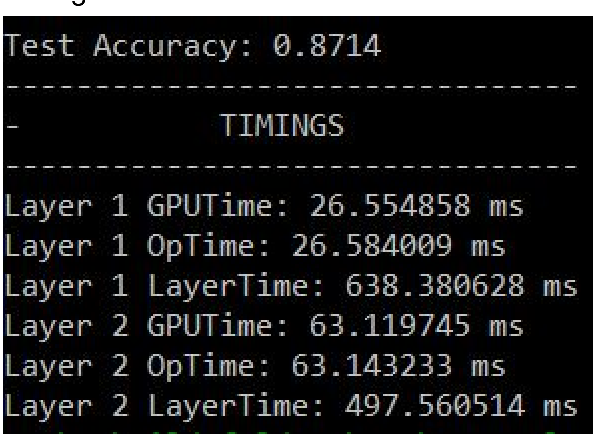
We had the idea of implementing this optimization from looking through the list of possible optimizations on the documentation. We expected that this approach would be effective because we haven't fiddled around with the block sizes at all up to this point, and thought doing so (in addition with thread coarsening) would yield a beneficial result.

Timings based on milestone 4 code:



```
Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 27.200195 ms
Layer 1 OpTime: 27.239074 ms
Layer 1 LayerTime: 686.241985 ms
Layer 2 GPUTime: 116.891079 ms
Layer 2 OpTime: 116.917959 ms
Layer 2 LayerTime: 569.835636 ms
```

Timings based on milestone 4 code with thread coarsening and block size optimization:



```
Test Accuracy: 0.8714
--------------------------------
-           TIMINGS
--------------------------------
Layer 1 GPUTime: 26.554858 ms
Layer 1 OpTime: 26.584009 ms
Layer 1 LayerTime: 638.380628 ms
Layer 2 GPUTime: 63.119745 ms
Layer 2 OpTime: 63.143233 ms
Layer 2 LayerTime: 497.560514 ms
```

CUDA Kernel statistics before applying this optimization:

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0       147930892           2      73965446.0        27212334       120718558  conv_forward_kernel
    0.0            2944           2          1472.0            1376            1568  do_not_remove_this_kernel
    0.0            2688           2          1344.0            1312            1376  prefn_marker_kernel
```

CUDA Kernel statistics after applying this optimization:

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------
  100.0        90457753           2      45228876.5        23692975        66764778  conv_forward_kernel
    0.0            5824           2          2912.0            2752            3072  prefn_marker_kernel
    0.0            2816           2          1408.0            1376            1440  do_not_remove_this_kernel
```

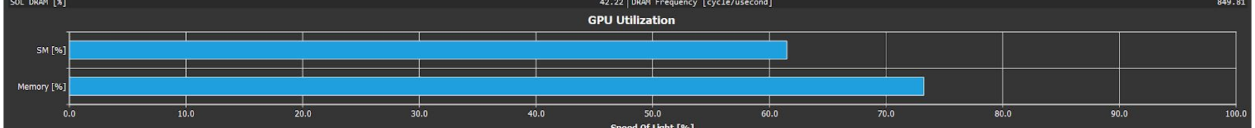
As you can see, the total, average, and minimum time using our conv_forward_kernel are lower after applying this optimization to the milestone 4 implementation.

Here is the Nsight Compute for further reference:

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 [%]	61.50	Duration [msecond]	26.60
SOL Memory [%]	73.24	Elapsed Cycles [cycle]	32,036,726
SOL L1/TEX Cache [%]	73.24	SM Active Cycles [cycle]	32,031,702.85
SOL L2 Cache [%]	16.51	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	42.22	DRAM Frequency [cycle/usecond]	849.81

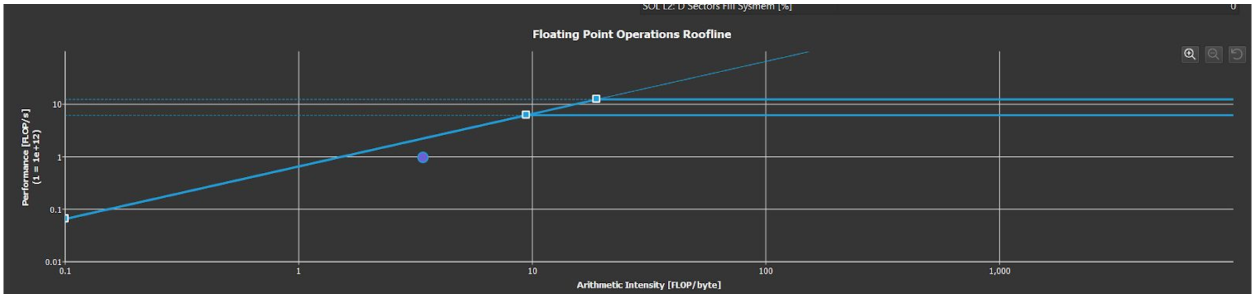


SOL SM Breakdown

SOL SM: Issue Active [%]	61.50
SOL SM: Inst Executed [%]	61.50
SOL SM: Pipe Fma Cycles Active [%]	49.17
SOL SM: Pipe Alu Cycles Active [%]	46.52
SOL SM: Inst Executed Pipe Adu [%]	33.73
SOL IDC: Request Cycles Active [%]	30.60
SOL SM: Inst Executed Pipe Lsu [%]	26.85
SOL SM: Mio2rf Writeback Active [%]	23.22
SOL SM: Mio Inst Issued [%]	21.86
SOL SM: Mio Pq Read Cycles Active [%]	16.61
SOL SM: Mio Pq Write Cycles Active [%]	15.93
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	4.37
SOL SM: Inst Executed Pipe Xu [%]	3.75
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	73.24
SOL L1: Lsu Writeback Active [%]	61.90
SOL GPU: Dram Throughput [%]	42.22
SOL L1: Lsuin Requests [%]	26.85
SOL L2: Xbar2lts Cycles Active [%]	16.51
SOL L2: T Sectors [%]	14.27
SOL L2: D Sectors Fill Device [%]	13.49
SOL L2: T Tag Requests [%]	12.14
SOL L2: Lts2xbar Cycles Active [%]	9.89
SOL L1: M L1tex2xbar Req Cycles Active [%]	9.42
SOL L2: D Sectors [%]	7.49
SOL L1: Data Bank Reads [%]	6.17
SOL L1: M Xbar2l1tex Read Sectors [%]	5.64
SOL L1: Data Bank Writes [%]	0.86
SOL L1: F Wavefronts [%]	0.00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0



External references: https://dl.acm.org/doi/pdf/10.1145/3194242

3. Two kernel implementations for different layer sizes

Motivation

While attempting to implement reduction across input channels within a block, we had to reduce the tiling width to remain under the thread limit for each block. We noticed that performance improved with smaller tiles, and measured the time for various block sizes against the base submission from PM4, taking the minimum Op times over three runs. The smaller layer, with one input channel, performed better with tiles of width 16, while the larger layer performed better with tiles of width 8. The obvious way to exploit this seemed to be to treat the single channel case as a special case, using a tile width of 16 and writing a kernel that eliminated the code required to handle multiple channels. Convolutions with multiple input channels used a tile width of 8.

Effect

The performance benefit appears to have been marginal over the base implementation with a tile width of 16. This was the timing output from one of our measurements:

```
--------------------------------
-          TIMINGS
--------------------------------
Layer 1 GPUTime: 17.898761 ms
Layer 1 OpTime: 17.923081 ms
Layer 1 LayerTime: 617.52697 ms
Layer 2 GPUTime: 55.093589 ms
Layer 2 OpTime: 55.122356 ms
Layer 2 LayerTime: 518.597423 ms
```

We also provide a table of timing measurements from the base implementation:

Block Size	Layer 1 Time (Minimum of 3)	Layer 2 Time (Minimum of 3)
8x4	30.99	65
8x8	23.48	**51.61**
16x16	**17.92**	55.90

Hybrid (This optimization)	**17.92**	55.12

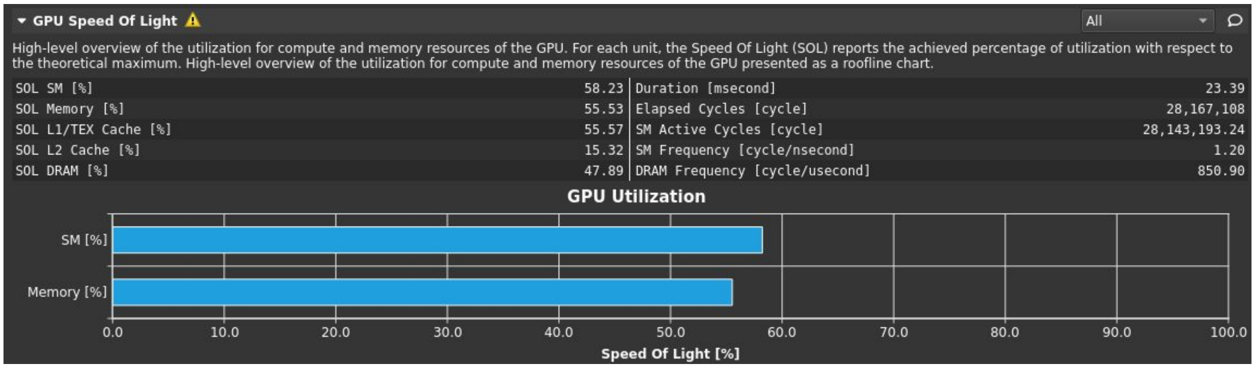
It is possible that the measurement from the 8x8 case was an outlier and that a tile width of 8 does not normally perform better for layer 2. Our two-kernel implementation, however, at least matches the best baseline performance, and has access to any benefit that a tile width of 8 has for the performance on multiple input channels.

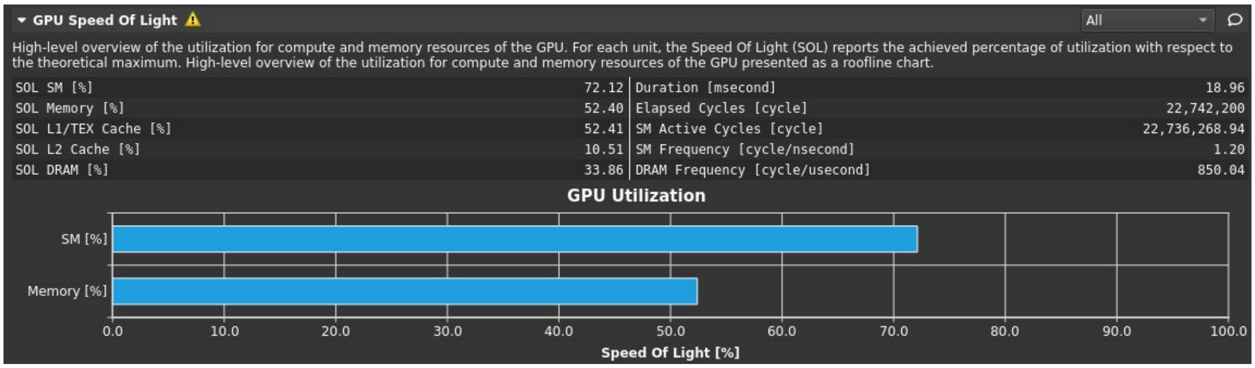
Nsys Output

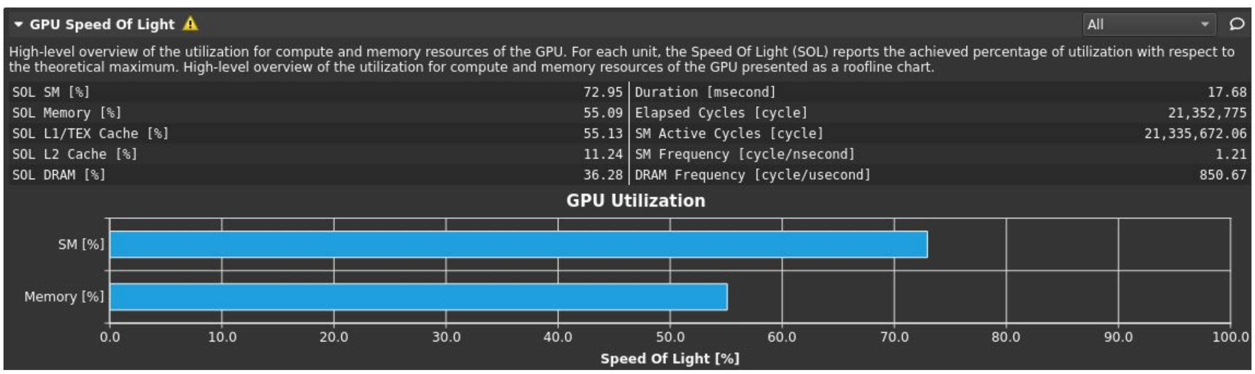
```
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   80.0      1113727705           6     185621284.2           16123       575747388  cudaMemcpy
   18.7       260156502           8      32519562.7           58602       251424688  cudaMalloc
    1.2        16080994           6       2680165.7           15805        15964591  cudaLaunchKernel
    0.2         2261191           8        282648.9           54754          817778  cudaFree
    0.0          349850           2        174925.0          172553          177297  cudaMemcpyToSymbol
    0.0           17141           4          4285.2            2435            7462  cudaDeviceSynchronize
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   75.4        55285311           1      55285311.0        55285311        55285311  conv_forward_kernel
   24.6        18065741           1      18065741.0        18065741        18065741  conv_forward_kernel_onechannel
    0.0            3872           2          1936.0            1312            2560  prefn_marker_kernel
    0.0            2720           2          1360.0            1280            1440  do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   91.3       945911286           2     472955643.0       389012847       556898439  [CUDA memcpy DtoH]
    8.7        89667333           6      14944555.5            1472        48032463  [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]
```

Analysis

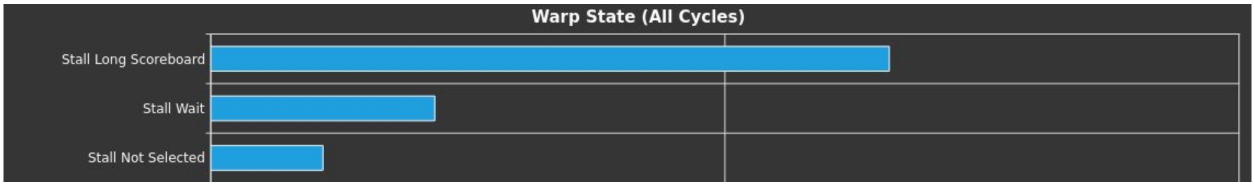
Nsight was used to try to make sense of these results. We first looked at the speed of light statistics for the 8x8, 16x16, and hybrid implementations, on the first layer. The hybrid and 16x16 implementations had significantly higher SM utilization than with a block width of 8:

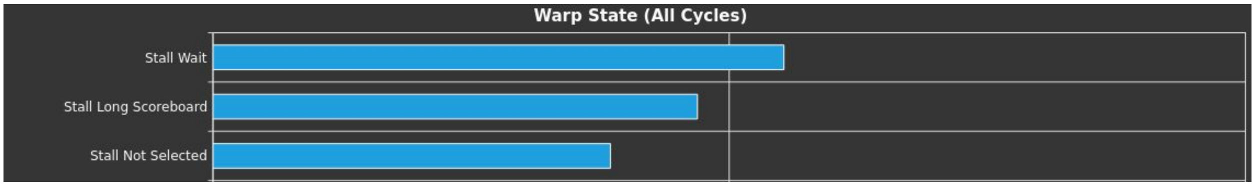


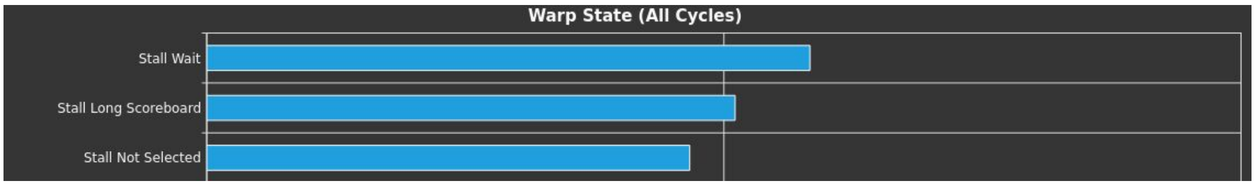




We further saw that that the 8x8 implementation stalled much more on scoreboard dependencies:

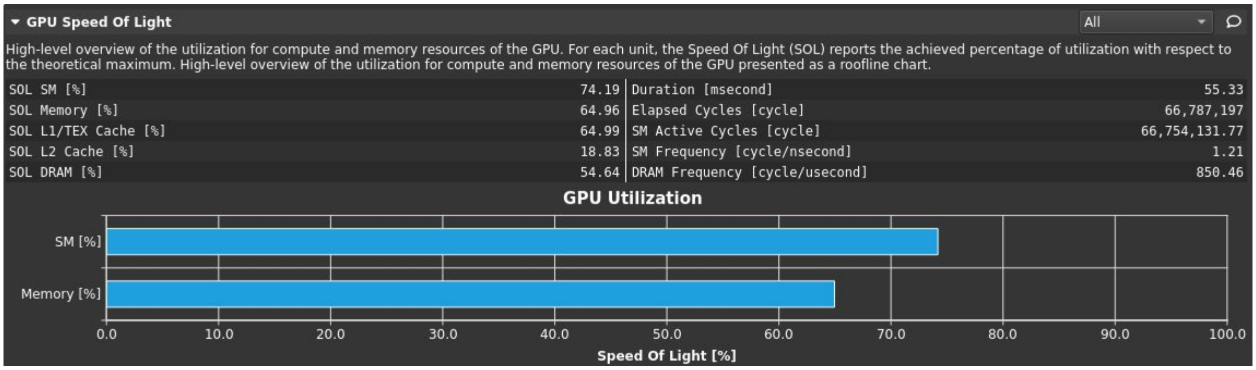


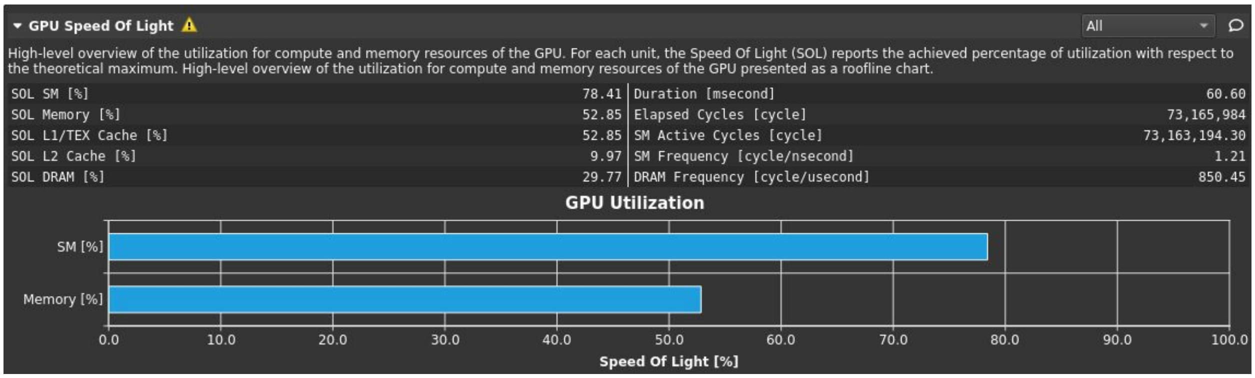


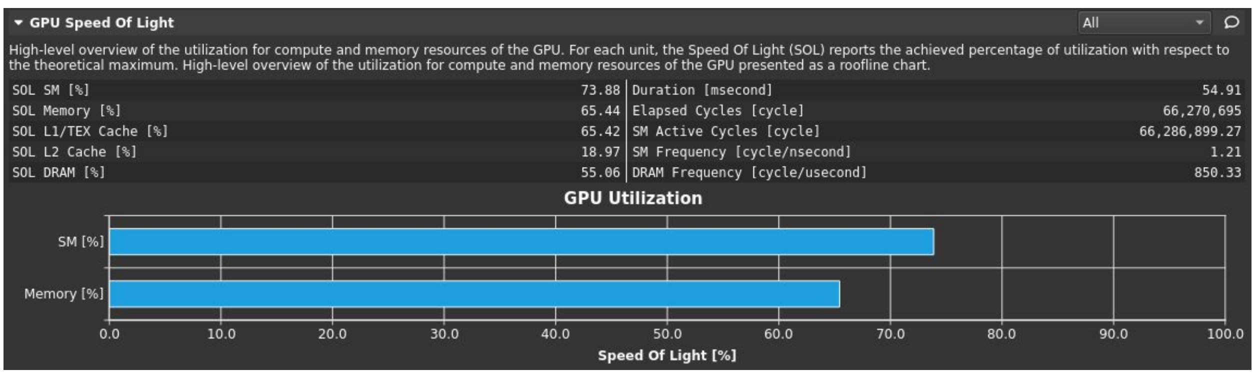


From these statistics we concluded that increasing the block width from 8 to 16 improved data reuse when accessing the L1 cache, which would have been used heavily because our base implementation does not use shared memory. This would not have affected invocations on the second layer as much, because it has data reuse across input channels as well as spatially in a particular input channel.

We had more trouble analyzing the performance behavior of the second kernel invocation, with the multi-channel input:







A higher memory utilization for 8x8 tiling is expected, as there is less data reuse within each tile. The occupancy statistics show that occupancy was a little higher for the kernel invocations that used 8x8 tiling, so it’s possible that a higher block limit made more warps available to hide latency over:

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	16
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	47.08	Block Limit Warps [block]	32
Achieved Active Warps Per SM [warp]	30.13	Block Limit SM [block]	32

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	44.02	Block Limit Warps [block]	8
Achieved Active Warps Per SM [warp]	28.17	Block Limit SM [block]	32

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	16
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	47.01	Block Limit Warps [block]	32
Achieved Active Warps Per SM [warp]	30.09	Block Limit SM [block]	32

References

N/A

Contributions (Final Milestone)

- Benoit - FP16
- Steven - Parameter sweeping / thread coarsening
- Leon - Two-kernel optimization.