ECE408 Applied Parallel Programming

Final Report

Prerit Oberai

Team Information

Team Member Names	Prerit Oberai, Jeonghyun Woo, Sagar Vyas
NetIDs	poberai2, jwoo15, sagarv2
Team Name	Team Ampere
School Affiliation	Illinois

Table 1: Team Information

Milestone 1

- Team registered on Google Sheet, nothing needed here

Milestone 2 - CPU Convolution

- RAI Output for running Mini-DNN on the CPU (CPU convolution implemented) for batch size of 10k images

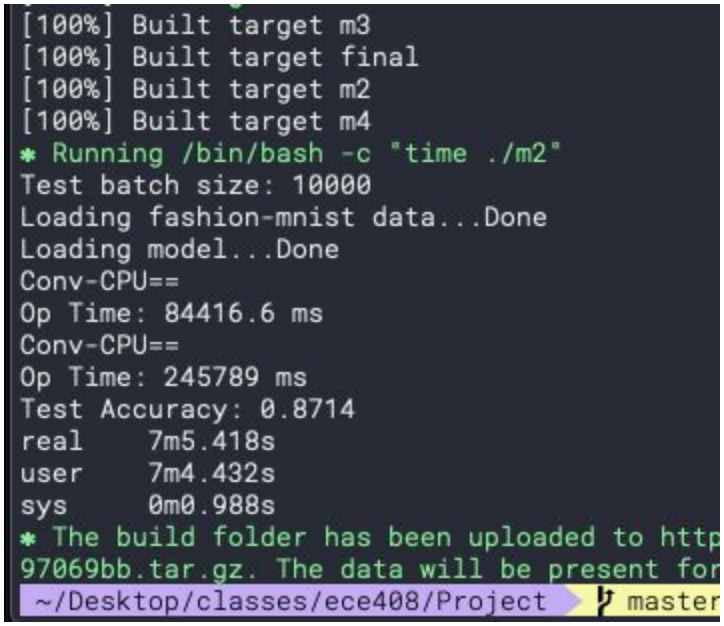


Figure 1: RAI output for Mini-DNN on CPU with CPU Convolution

- List Op Times (CPU convolution implemented) for batch size of 10k images

Batch Size	**Operation Execution Time**
100	822.935 ms + 2458.4 ms
1000	8195.81 ms + 23951.8 ms
10000	84416.6 ms+ 245789 ms

Table 2: Operation Times for CPU Convolution

- List whole program execution time (CPU convolution implemented) for batch size of 10k images

Batch Size	**Program Execution Time**
100	real 0m4.248s user 0m4.228s sys 0m0.020s
1000	real 0m41.584s user 0m41.463s sys 0m0.120s
10000	real 7m5.418s user 7m4.432s sys 0m0.988s

Table 3: Entire Program Execution Times for CPU Convolution

Milestone 3 - GPU Convolution

After implementing the GPU version of the convolution, for the default batch size (10000), I get the following output showing that the GPU version of the convolution is implemented correctly because I get the same test accuracy compared to the CPU implementation.

```
* Running /bin/bash -c "./m3"
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Allocating Memory..
Settting Kernel Dimensions
Launching Kernel
Copying Data back to Host
Freeing Memory
Op Time: 839.665 ms
Conv-GPU==
Allocating Memory..
Settting Kernel Dimensions
Launching Kernel
Copying Data back to Host
Freeing Memory
Op Time: 505.978 ms
Test Accuracy: 0.8714
```

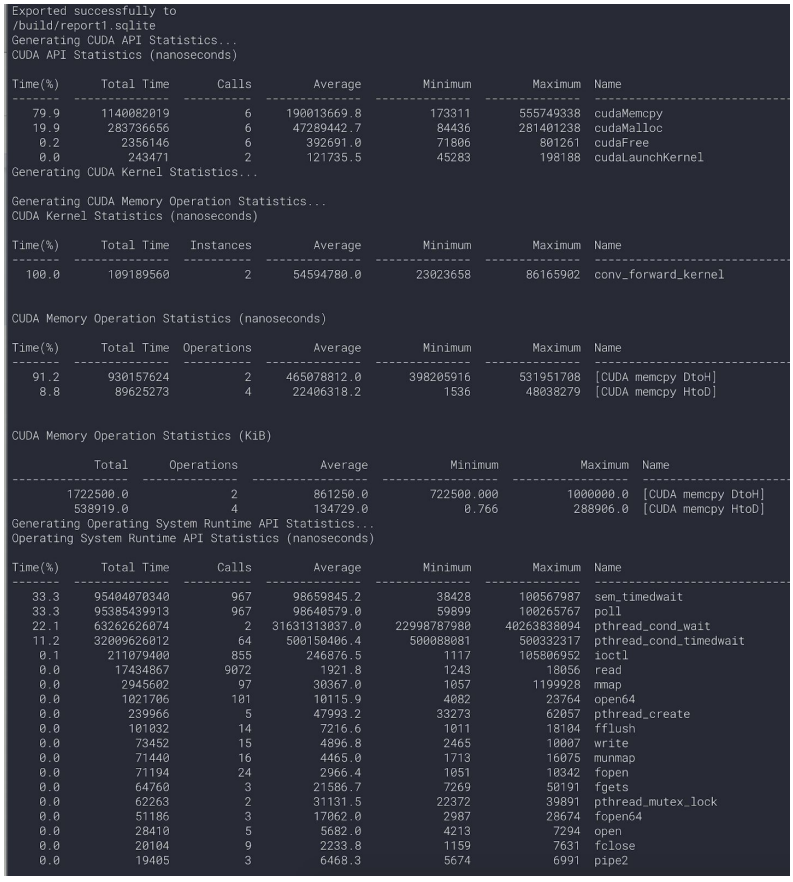
Figure 2: Figure 1: RAI output for Mini-DNN on CPU with GPU Convolution

Specifically, the timing and accuracy that I get from my implementation for the batch sizes of 100, 1000, and 10000 are included in the following table.

Batch Size	**Operation Execution Time**	**Accuracy**
100	91.3695 ms + 5.75664 ms	0.86
1000	130.565 ms + 50.7728 ms	0.886
10000	839.665 ms + 505.978 ms	0.8714

Table 4: Timing and Accuracy for GPU convolution with varying Batch Size

And then after modifying `rai_build.yml` to generate a nsys profile instead of just executing the code, as per the instructions, I get the following output showing the detailed statistics from running my code



```
Exported successfully to
/build/report1.sqlite
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   79.9      1140082019           6     190013669.8          173311       555749338  cudaMemcpy
   19.9       283736656           6      47289442.7           84436       281401238  cudaMalloc
    0.2         2356146           6        392691.0           71806          801261  cudaFree
    0.0          243471           2        121735.5           45283          198188  cudaLaunchKernel
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0       109189560           2      54594780.0        23023658        86165902  conv_forward_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   91.2       930157624           2     465078812.0       398205916       531951708  [CUDA memcpy DtoH]
    8.8        89625273           4      22406318.2            1536        48038279  [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     95404070340         967     98659845.2           38428       100567987  sem_timedwait
   33.3     95385439913         967     98640579.0           59899       100265767  poll
   22.1     63262626074           2  31631313037.0     22998787980     40263838094  pthread_cond_wait
   11.2     32009626012          64    500150406.4       500088081       500332317  pthread_cond_timedwait
    0.1       211079400         855       246876.5            1117       105806952  ioctl
    0.0        17434867        9072         1921.8            1243           18056  read
    0.0         2945602          97        30367.0            1057         1199928  mmap
    0.0         1021706         101        10115.9            4082           23764  open64
    0.0          239966           5        47993.2           33273           62057  pthread_create
    0.0          101032          14         7216.6            1011           18104  fflush
    0.0           73452          15         4896.8            2465           10007  write
    0.0           71440          16         4465.0            1713           16075  munmap
    0.0           71194          24         2966.4            1051           10342  fopen
    0.0           64760           3        21586.7            7269           50191  fgets
    0.0           62263           2        31131.5           22372           39891  pthread_mutex_lock
    0.0           51186           3        17062.0            2987           28674  fopen64
    0.0           28410           5         5682.0            4213            7294  open
    0.0           20104           9         2233.8            1159            7631  fclose
    0.0           19405           3         6468.3            5674            6991  pipe2
```

Figure 3: Detailed Stats for GPU Convolution

As we can see from the figure above, the list of kernels that collectively consume more than 90% of the program time are: `conv_forward_kernel`.

On the other hand, the list of CUDA API calls that collectively consume more than 90% of the program time are: `cudaMemcpy` and `cudaMalloc`. The reason `cudaMalloc` is included in this list is that each call consumes 19.9% but because it's called 6 times, that collectively consumes more than 90%.

The key differences between kernels and API calls are that kernels allow programmers to define their own functions when called, are executed in parallel by different threads. API calls, on the other hand, are made by the code (or other CUDA API calls in the code) into the CUDA driver and/or runtime libraries. The driver API provides an additional level of control by exposing even lower-level concepts such as contexts and modules.

Finally, the execution of my kernel of the GPU SOL utilization looks like the following:

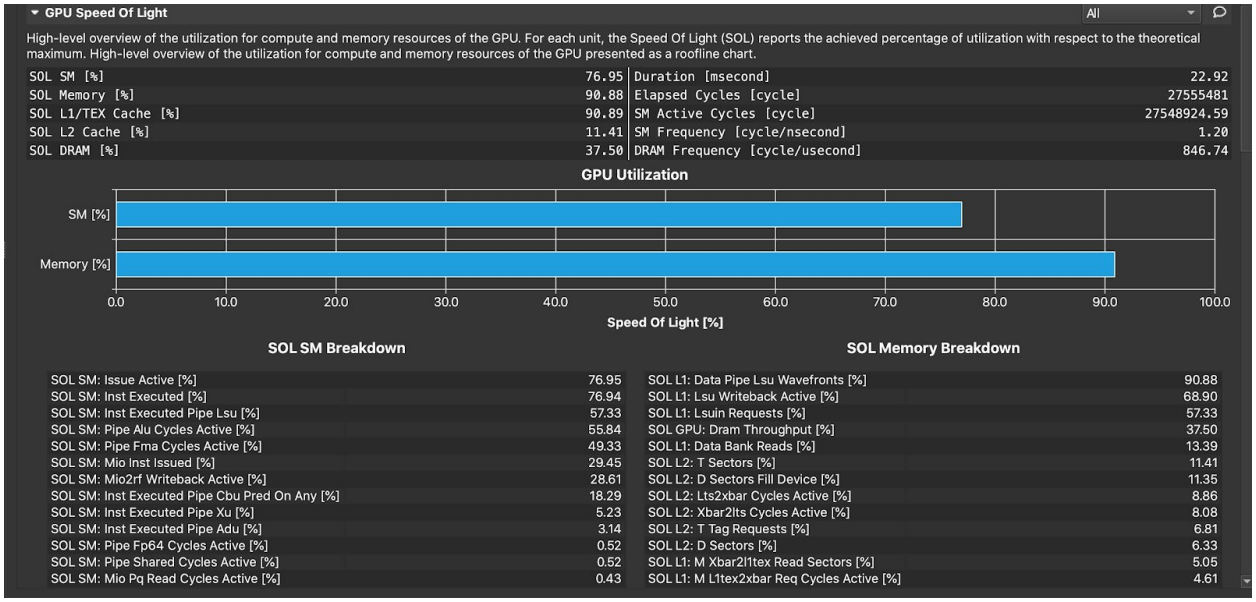


Figure 4: Nsight Stats for GPU Baseline Convolution for batch size = 10000

Milestone 4.1 - Add 3 Optimizations

Optimization 1: Using Fixed Point Arithmetic

Our first optimization was using fixed point arithmetic instead of floating point and the way we arrived at this optimization opportunity was due to the recommendations offered by the instructors on the project github.

We thought that this approach would be fruitful because floating point operations are inherently time-consuming for a generic processor as it involves multiple steps to account for the

difference in the exponential values. Fixed point operations, however mean that the decimal point is fixed in the number representation allowing for the operation hardware to be simplified. [1,2] Additionally, having done more research into why fixed point arithmetic on GPUs might offer benefits, we arrived at a blog post indicating that using fixed point numbers can “reduce the amount of data that needs to move from device memory to the GPU” [3] and decided that it was enough reason to try it out.

Ultimately, after implementing fixed point implementation [4,5], we observed that the test accuracy hadn’t changed (which was what we wanted) but the GPU time, operation time and layer times varied slightly and were actually worse in certain situations as shown in table 5.

Batch Size	**Baseline Operation Times**	**FP16 Operation Times**
100	Layer 1 GPUTime: 0.150943 ms Layer 1 OpTime: 0.166175 ms Layer 1 LayerTime: 11.281809 ms Layer 2 GPUTime: 0.786906 ms Layer 2 OpTime: 0.80297 ms Layer 2 LayerTime: 6.142256 ms	Layer 1 GPUTime: 0.303806 ms Layer 1 OpTime: 0.331198 ms Layer 1 LayerTime: 7.946445 ms Layer 2 GPUTime: 1.139608 ms Layer 2 OpTime: 1.159576 ms Layer 2 LayerTime: 6.317432 ms
1000	Layer 1 GPUTime: 1.620341 ms Layer 1 OpTime: 1.636309 ms Layer 1 LayerTime: 67.498953 ms Layer 2 GPUTime: 8.592806 ms Layer 2 OpTime: 8.616421 ms Layer 2 LayerTime: 61.967887 ms	Layer 1 GPUTime: 2.982319 ms Layer 1 OpTime: 3.041807 ms Layer 1 LayerTime: 200.7771 ms Layer 2 GPUTime: 11.345533 ms Layer 2 OpTime: 11.386781 ms Layer 2 LayerTime: 61.598933 ms
10000	Layer 1 GPUTime: 33.531234 ms Layer 1 OpTime: 33.541186 ms Layer 1 LayerTime: 657.95256 ms Layer 2 GPUTime: 86.001188 ms Layer 2 OpTime: 86.027556 ms Layer 2 LayerTime: 581.144942 ms	Layer 1 GPUTime: 29.563294 ms Layer 1 OpTime: 29.600254 ms Layer 1 LayerTime: 638.703443 ms Layer 2 GPUTime: 112.910752 ms Layer 2 OpTime: 112.9504 ms Layer 2 LayerTime: 581.635624 ms

Table 5: Timing comparing GPU convolution with with baseline kernel vs FP16 kernel

Because this was not as insightful as we thought it would be, we then generated an nsight report outlining the memory and compute usage. Having done so, we arrived at the results shown in Figure 5. As it can be seen when comparing Figures 4 and 5, this optimization was fruitful because the memory utilization drastically decreased from ~90% down to 70%. Therefore, whereas in the baseline kernel, we were bounded by the memory utilization, we are now bounded by the compute (SM) utilization, illustrating that this optimization was indeed beneficial.

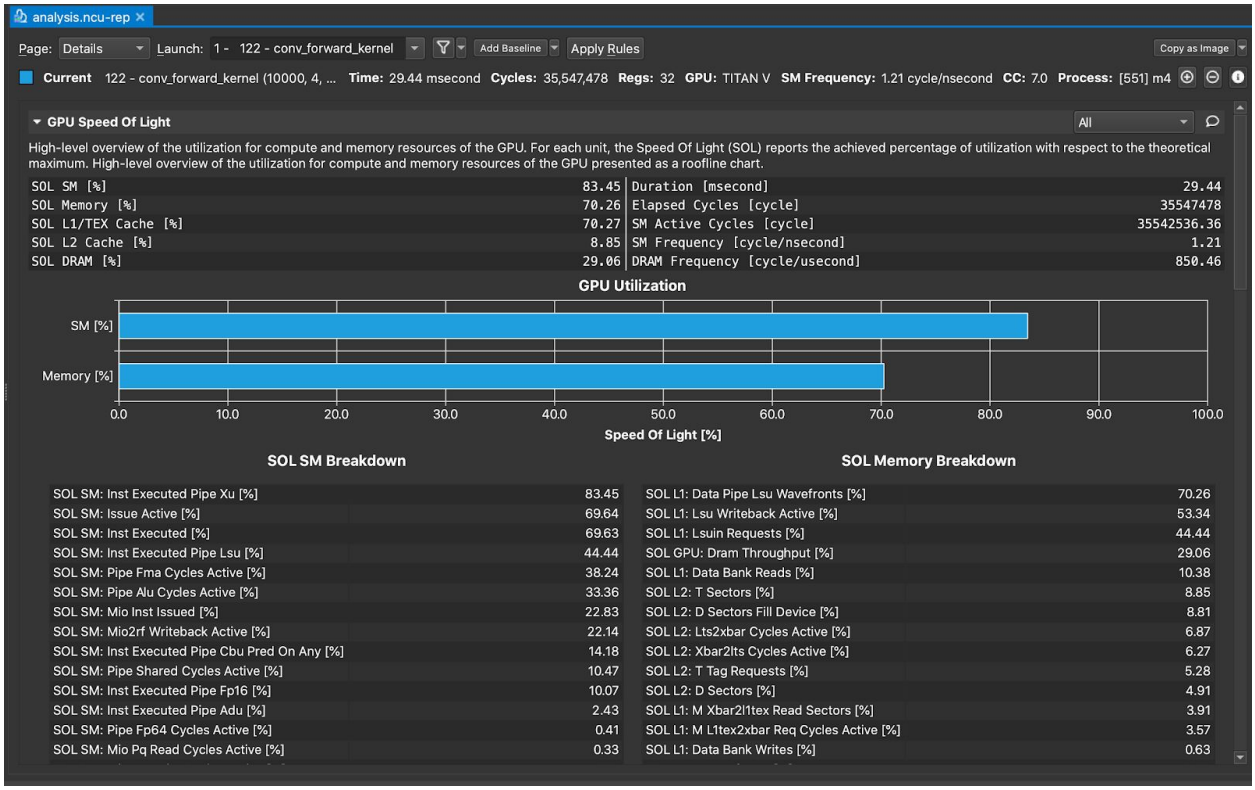


Figure 5: Nsight Stats for GPU Convolution with FP16 Optimization for batch size = 10000

Optimization 2: Using Constant Memory for Weights + Varying Tile Size

For our 2nd optimization we loaded the convolution weights in constant memory. In addition to that, we also explored the effect of the parameter TILE_WIDTH on the operation times. Four different values of TILE_WIDTH were used viz., 4, 8, 16 and 32. The results of this optimization (for a batch size of 10000) are presented in Table 6 shown below.

Type	TILE_WIDTH	Operations times
Baseline	32	Layer 1 GPUTime: 33.531234 ms Layer 1 OpTime: 33.541186 ms Layer 1 LayerTime: 657.95256 ms Layer 2 GPUTime: 86.001188 ms Layer 2 OpTime: 86.027556 ms Layer 2 LayerTime: 581.144942 ms
Constant Memory	4	Layer 1 GPUTime: 66.907754 ms Layer 1 OpTime: 66.925994 ms Layer 1 LayerTime: 672.242977 ms Layer 2 GPUTime: 179.527176 ms Layer 2 OpTime: 179.550504 ms

		Layer 2 LayerTime: 615.589251 ms
Constant Memory	8	Layer 1 GPUTime: 28.894742 ms Layer 1 OpTime: 28.91135 ms Layer 1 LayerTime: 643.301828 ms Layer 2 GPUTime: 66.781428 ms Layer 2 OpTime: 66.8101 ms Layer 2 LayerTime: 508.12481 ms
Constant Memory	16	Layer 1 GPUTime: 14.718295 ms Layer 1 OpTime: 14.735703 ms Layer 1 LayerTime: 605.94236 ms Layer 2 GPUTime: 77.13176 ms Layer 2 OpTime: 469.649213 ms Layer 2 LayerTime: 508.131762 ms
Constant Memory	32	Layer 1 GPUTime: 23.131365 ms Layer 1 OpTime: 23.154245 ms Layer 1 LayerTime: 636.520045 ms Layer 2 GPUTime: 140.392697 ms Layer 2 OpTime: 140.418937 ms Layer 2 LayerTime: 569.271794 ms

Table 6: Timing comparing constant memory GPU convolution for 4 different TILE_WIDTH values with baseline kernel

As seen in Table 6, a TILE_WIDTH of 16 works best for the constant memory optimization for the 1st layer whereas, a TILE_WIDTH of 8 works best for the 2nd layer. This opens up a new avenue of optimization where we can implement a different TILE_WIDTH for the first and second convolution layer in our convolutional network. Overall, a TILE_WIDTH of 8 works best as the total operation time is approximately 96 ms i.e., almost 24 ms lower than the baseline.

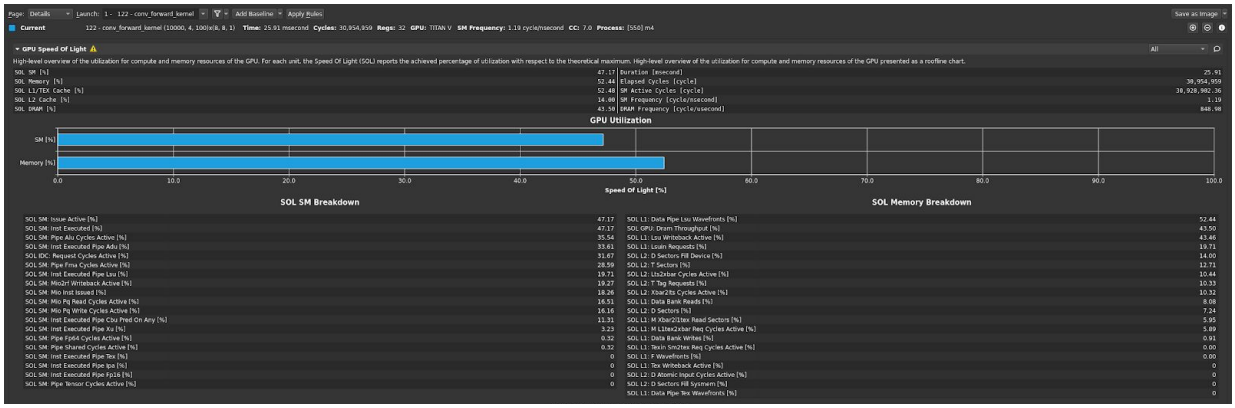


Figure 6: Nsight Stats for constant memory GPU Convolution with TILE_WIDTH = 8 for batch size = 10000

To gain a better understanding of why a TILE_WIDTH of 8 performs better than a TILE_WIDTH of 32, we profiled our code and generated Fig. 6 using Nsight. As seen there, the SM and memory utilizations for the constant memory optimization are quite low compared to the baseline run shown in Fig. 4. The lower SM utilization can be attributed to the lower value of TILE_WIDTH (8) for the constant memory optimization as opposed to the greater value (32) for the baseline, whereas the lower memory utilization is a direct outcome of loading the weight in constant memory. From these observations, it is clear that the performance boost is strictly a result of minimizing calls to global memory by storing the convolution weights in constant memory.

Optimization 3: Unroll + Shared Memory Matrix Multiplication

Our third optimization was using unrolling and shared memory matrix multiplication techniques. We thought this optimization would be beneficial because we could get a speed up due to adapting several optimized matrix multiplications even though additional time would be spent on unrolling the input and weights [5]. However, we got a significant speed drop as shown in table 7. We think there are three reasons for speed down.

In order to implement this optimization, we needed to split the input into mini batches since our global memory capacity (11.78GB) was not enough to contain 10K batches of unrolling input images. This meant that we faced a significant bottleneck because the kernels needed to run in an iterative manner. As a further optimization, we believe this could be accelerated by using fixed point arithmetic (optimization 1) because we could reduce the computation time when unrolling the input images and can compute the output with only one kernel.

Second, since we were using different kernels for unrolling and shared-memory matrix multiply, we face another performance overhead. Specifically, we see an additional 10.21 us for unrolling the input and 5.41 us for reshaping at the end which is required to run matrix multiplication kernel - which only takes 10.05 us (figures 7- 9). We think we can get a greater speed up by fusing the kernels for unrolling and matrix multiplication.

Third, using a naive shared matrix multiplication implementation is not enough to efficiently utilize GPU resources. Figure 8 shows the penalty we pay with the compute and memory resources in GPU when using naive matrix multiplication. To achieve a greater speed up we will try to use a more advanced matrix multiplication algorithm for the final submission, such as register-tiled matrix multiplication.

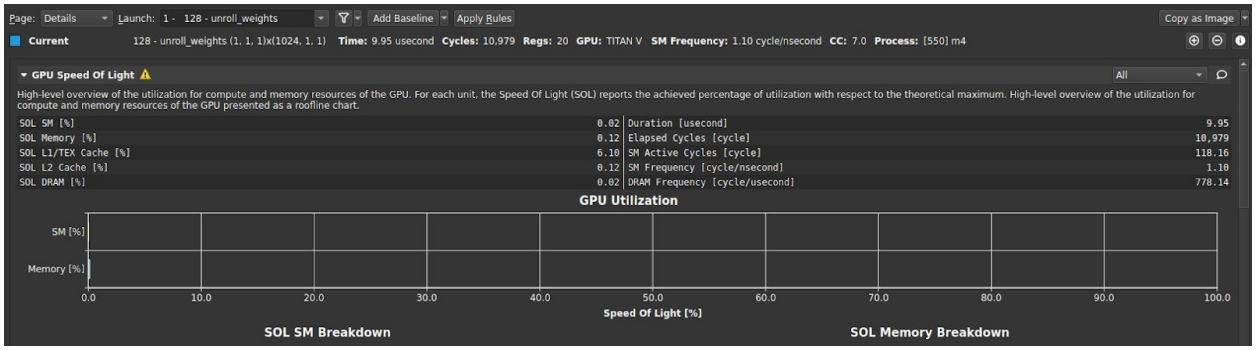


Figure 6: Nsight Stats for unrolling weights kernel

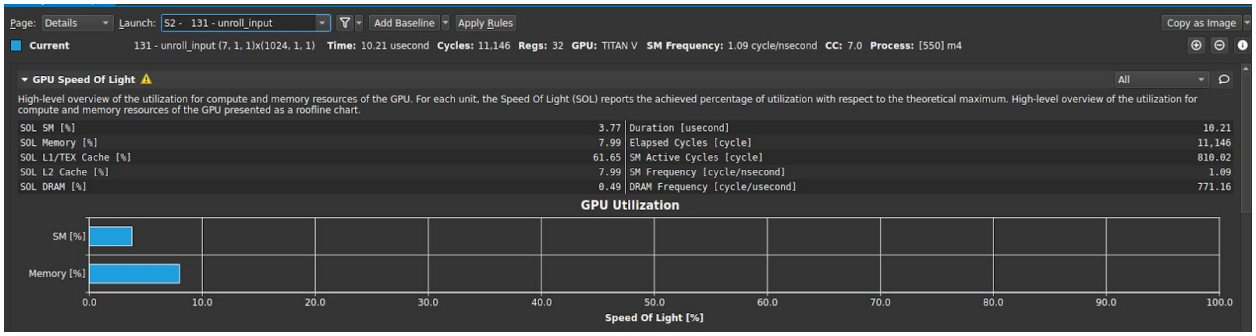


Figure 7: Nsight Stats for unrolling input kernel

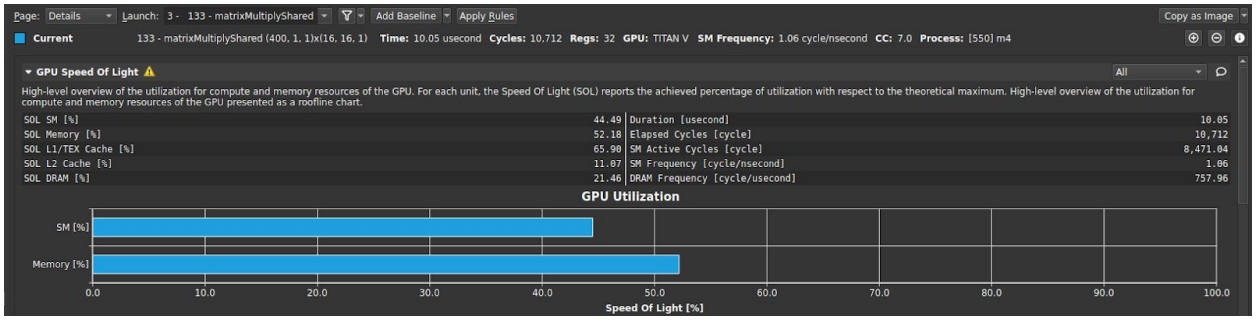


Figure 8: Nsight Stats for shared matrix multiply kernel

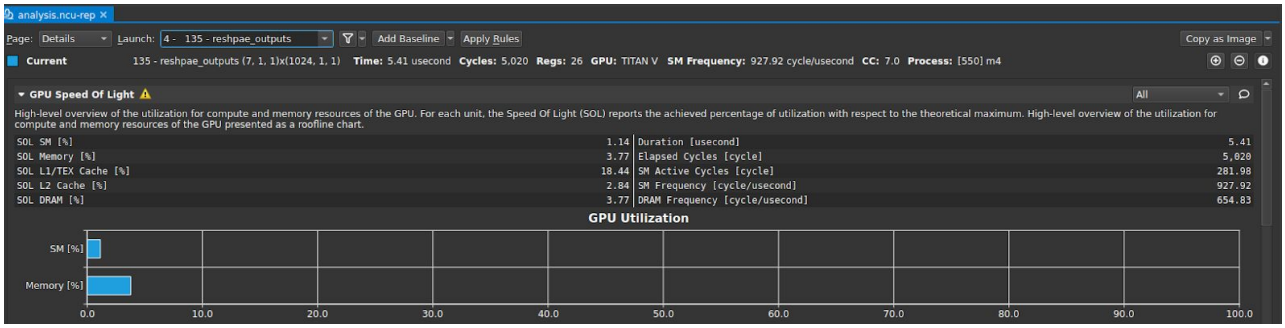


Figure 9: Nsight Stats for reshape kernel

Batch Size	**Baseline Operation Times**	**Unroll + Shared Memory Operation Times**
100	Layer 1 GPUTime: 0.150943 ms Layer 1 OpTime: 0.166175 ms Layer 1 LayerTime: 11.281809 ms Layer 2 GPUTime: 0.786906 ms Layer 2 OpTime: 0.80297 ms Layer 2 LayerTime: 6.142256 ms	Layer 1 GPUTime: 1.850167 ms Layer 1 OpTime: 2.158643 ms Layer 1 LayerTime: 8.906112 ms Layer 2 GPUTime: 2.590801 ms Layer 2 OpTime: 3.508012 ms Layer 2 LayerTime: 15.348165 ms
1000	Layer 1 GPUTime: 1.620341 ms Layer 1 OpTime: 1.636309 ms	Layer 1 GPUTime: 18.493263 ms Layer 1 OpTime: 31.206955 ms

	Layer 1 LayerTime: 67.498953 ms Layer 2 GPUTime: 8.592806 ms Layer 2 OpTime: 8.616421 ms Layer 2 LayerTime: 61.967887 ms	Layer 1 LayerTime: 100.196876 ms Layer 2 GPUTime: 26.111311 ms Layer 2 OpTime: 29.158747 ms Layer 2 LayerTime: 75.945948 ms
10000	Layer 1 GPUTime: 33.531234 ms Layer 1 OpTime: 33.541186 ms Layer 1 LayerTime: 657.95256 ms Layer 2 GPUTime: 86.001188 ms Layer 2 OpTime: 86.027556 ms Layer 2 LayerTime: 581.144942 ms	Layer 1 GPUTime: 174.345829 ms Layer 1 OpTime: 201.846313 ms Layer 1 LayerTime: 786.21534 ms Layer 2 GPUTime: 233.728146 ms Layer 2 OpTime: 260.214675 ms Layer 2 LayerTime: 687.945379 ms

Table 7: Timing comparing GPU convolution with with baseline kernel vs Unroll + GEMM Optimization kernel

Team Organization + Work Splitting

For the purposes of Milestone 4, we decided to split up the work such that one team member would be responsible for a single optimization but could obviously reach out to others on the team to ask for help/ideas. The idea behind this method was that we couldn't actually meet up and work through the optimization together and so by splitting up the work evenly, each of us had a better understanding of our respective optimizations. For the final project, we're hoping to work more intimately and combine these optimizations in order to climb up the leaderboard!

References

[1] https://www.ti.com/lit/wp/spry061/spry061.pdf?ts=1605931009759
[2] https://www.microcontrollertips.com/difference-between-fixed-and-floating-point/
[3]https://stackoverflow.com/questions/35198856/half-precision-difference-between-float2half-vs-float2half-rn/35202978
[4] https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html
[5] Kim, H., Nam, H., Jung, W., & Lee, J. (2017, April). Performance analysis of CNN frameworks for GPUs. In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 55-64). IEEE.