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

Additional Note:

Please note that we moved milestone 5 to the top of the report as per piazza post @665. This however caused the figure numbers to be out of order and so Figure 1 (related to milestone 1) actually comes after Figure 10 (related to milestone 5). We hope that this doesn't cause any confusion as all the figures are referenced correctly in the report. Thank you for your efforts.

Milestone 5 - Add 3 More Optimizations

Optimization 4: Shared Input Memory + Constant Memory for Weights Convolution

For our fourth optimization, we decided to take advantage of shared and constant memory. We arrived at this optimization method through the recommendations posted on the project github as well as the lectures in class.

Notably, we thought this approach would be beneficial because of the clear advantages of using shared memory over global memory. For instance, NVIDIA claims that shared memory is 100x faster than accessing global memory [6] and we also saw the advantages of using shared memory for matrix multiplication in optimization 3. However, we hadn't tried loading the inputs to shared memory and conducting convolution yet and so for this optimization method, we did just that.

Because we saw significant performance results from using shared memory for inputs, we also tried loading the weights into shared memory but noticed very little improvement (in certain cases, we saw none). After further thinking through why this could be the case, we concluded that it was inefficient for each thread block to load in the same wights. Instead, it was more efficient to actually load the weights into constant memory (because they never changed)

and in this manner,all the thread blocks could have access to the same weights. Therefore having implemented both shared memory for the inputs and loading the weights to constant memory, we observed the results as presented in Table 8.

Batch Size	**Baseline Operation Times**	**Constant Memory (weights) + Shared Memory (input) 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: 0.179903 ms Layer 1 OpTime: 0.207934 ms Layer 1 LayerTime: 6.91153 ms Layer 2 GPUTime: 1.009783 ms Layer 2 OpTime: 1.040887 ms Layer 2 LayerTime: 6.149671 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: 1.70463 ms Layer 1 OpTime: 1.732022 ms Layer 1 LayerTime: 67.340219 ms Layer 2 GPUTime: 9.577317 ms Layer 2 OpTime: 9.608677 ms Layer 2 LayerTime: 57.871739 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: 16.832097 ms Layer 1 OpTime: 16.862721 ms Layer 1 LayerTime: 653.87165 ms Layer 2 GPUTime: 90.796628 ms Layer 2 OpTime: 90.848436 ms Layer 2 LayerTime: 547.930517 ms

Table 8: Timing comparing GPU convolution with with baseline kernel vs shared input memory and constant memory for weights

As it's shown in Table 8, this optimization method reduced our operation times by 15ms for the 10000 batch size. However, because operation times aren't insightful in how the kernel is running, we generated the following nsight report for our shared input memory + constant weight kernel as shown in Figure 10.

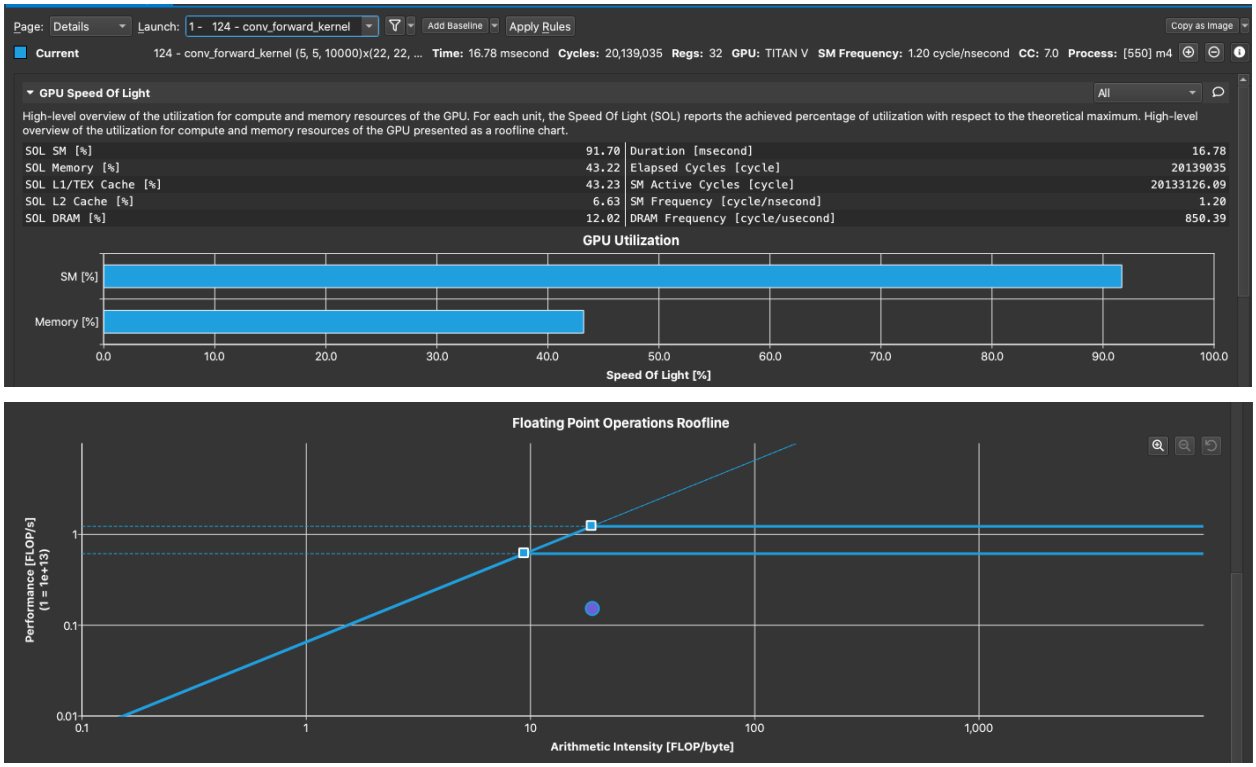


Figure 10: Nsight Stats for convolution with shared input memory + constant weights

Therefore, observe from Figure 10 that this optimization was fruitful in that we were able to further decrease memory utilization from ~70% (with previous optimization methods) to ~40%! Just like before, it is clearly seen that we are now bounded by SM utilization and not memory bandwidth, illustrating that this optimization was indeed beneficial,

Optimization 5: Kernel fusion for unrolling and matrix-multiplication

Our fifth optimization was implementing a fused kernel for unrolling and shared matrix multiplication. Based on our observations from optimization 3, we thought this method would be beneficial based on the insight that we could reduce extra computation time due to unrolling the input and reshaping the output. We thought we could achieve more performance improvement through this because we can compute the whole batch by launching only one kernel instead of launching multiple times. This is because we no longer needed to store the unrolling input function map in global memory which made us use mini batches.

Although our fused kernel approach gained almost 184 ms speed up in total operation times compared to the previous separate unroll and shared matrix multiplication kernel method (Optimization 3), this method was still about 158 ms slower than the baseline kernel, as shown in Table 9. Major speed up comes from integrating the unrolling kernel and shared matrix multiplication kernel, but we failed to get performance improvement by removing kernel iterations due to usage of mini batches. To gain an insight of the reason for this and speed down compared to the baseline kernel, we analyzed the kernel with Nsight as shown in Figure 11.

Batch Size	Baseline Operation Times (TILE_WIDTH = 32)	Separate Unroll + Shared Matrix Multiplication Operation TImes (Optimization 3) (TILE_WIDTH =16)	Constant Memory (weights) + Fused kernel Operation Times (TILE_WIDTH = 16)
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	Layer 1 GPUTime: 1.305591 ms Layer 1 OpTime: 1.323639 ms Layer 1 LayerTime: 8.272256 ms Layer 2 GPUTime: 1.523221 ms Layer 2 OpTime: 1.537941 ms Layer 2 LayerTime: 6.801638 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: 18.493263 ms Layer 1 OpTime: 31.206955 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	Layer 1 GPUTime: 12.97437 ms Layer 1 OpTime: 12.99853 ms Layer 1 LayerTime: 76.116892 ms Layer 2 GPUTime: 15.172306 ms Layer 2 OpTime: 15.19493 ms Layer 2 LayerTime: 59.507319 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	Layer 1 GPUTime: 121.613873 ms Layer 1 OpTime: 128.646129 ms Layer 1 LayerTime: 708.92201 ms Layer 2 GPUTime: 149.714864 ms Layer 2 OpTime: 149.741423 ms Layer 2 LayerTime:571.167739 ms

Table 9: Timing comparing GPU convolution with with baseline kernel vs Unroll + shared matrix multiplication kernel vs fused kernel

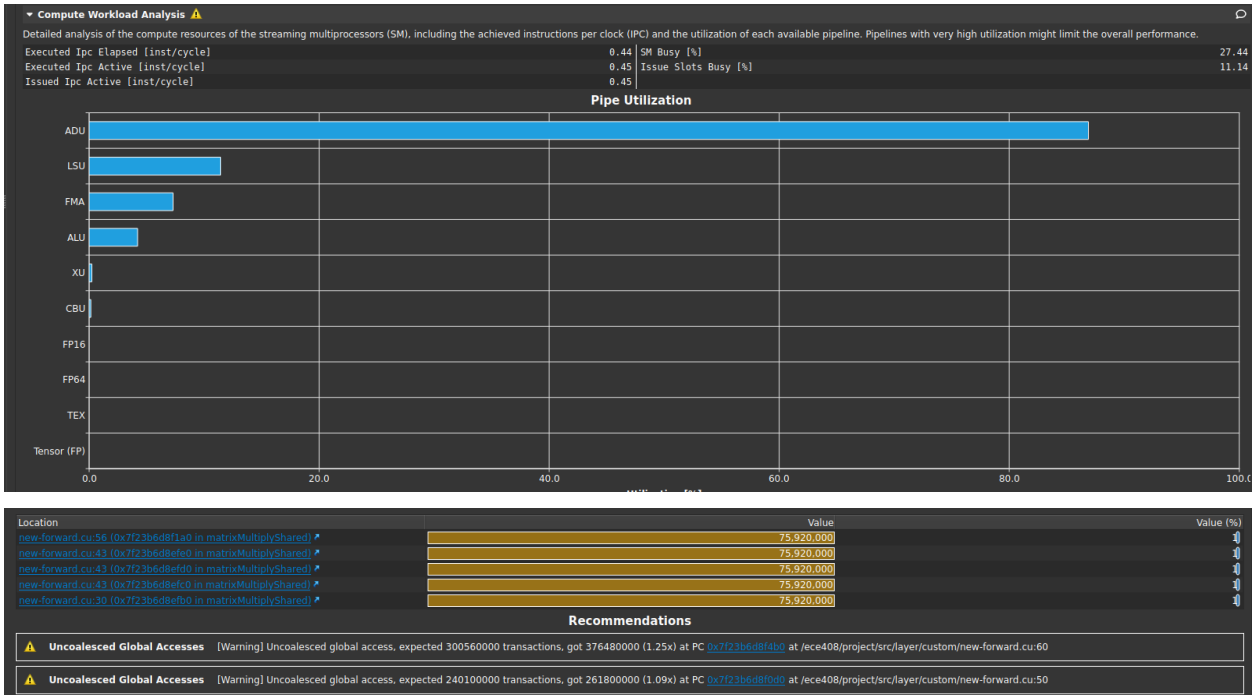


Figure11: Nsight Stats for fused kernel

Based on the profiling result from Nsight (Figure 11), we noticed three problems in the kernel. First, as shown in the second and third figure of Figure 11, although we achieved high parallelism in our kernel, this benefit was counterbalanced due to a lot of control divergence. Second, global memory accesses were not coalesced when we loaded the input data from global memory to shared memory and wrote from shared memory data into global memory. Third, each thread calculated each output element serially. We believe the first problem could be mitigated by reshaping our block shape. The second problem would be alleviated by changing an index to generate coalesced memory accesses. The last problem would be reduced by combining restrict and loop unrolling optimization methods. Unfortunately, we had not been able to try these strategies due to limited time.

Optimization 6: Varying Tile Size

For our sixth optimization, we studied the effect of varying TILE_WIDTH on Optimizations 5 and 6. We believed this would help because it would address taking advantage of parallelism as well as limiting control divergence. The values of TILE_WIDTH chosen for this optimization were 4, 8, 16 and 32. Table 9 shows the results for this optimization as follows: (Note that all the results shown are for a batch size of 10000 images.)

TILE_WIDTH	Type	Total OP time (1st layer + 2nd layer)
4	Constant Memory	246.476 ms
	Shared input + constant	-

	weights (Optimization 4)	
	Kernel fusion (Optimization 5)	809.255 ms
8	Constant Memory	95.721 ms
	Shared input + constant weights (Optimization 4)	191.815 ms
	Kernel fusion (Optimization 5)	535.568 ms
16	Constant Memory	484.385 ms
	Shared input + constant weights (Optimization 4)	81.261 ms
	Kernel fusion (Optimization 5)	273.597 ms
32	Constant Memory	163.573 ms
	Shared input + constant weights (Optimization 4)	-
	Kernel fusion (Optimization 5)	272.833 ms

Table 10: Op time comparison for optimizations 4, 5 and 2. The best time is achieved for optimization 4 using a TILE_WIDTH of 16.

As seen in Table 9, loading the input into the shared memory was beneficial to reducing the Op time as accessing data from the shared memory is much faster than accessing data from global memory. For additional insight, we generated Nsight reports for the case of optimization 4 with a TILE_WIDTH of 16 (new best) and optimization 2 with a TILE_WIDTH of 8 (previous best). The Nsight reports for optimization 2 and 4 are shown in Figures 11 and 12, respectively.

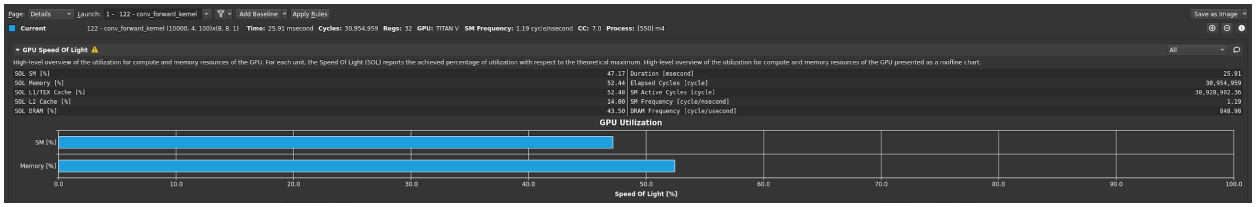


Fig 12. Nsight GPU SOL results for optimization 2 with a TILE_WIDTH of 8.

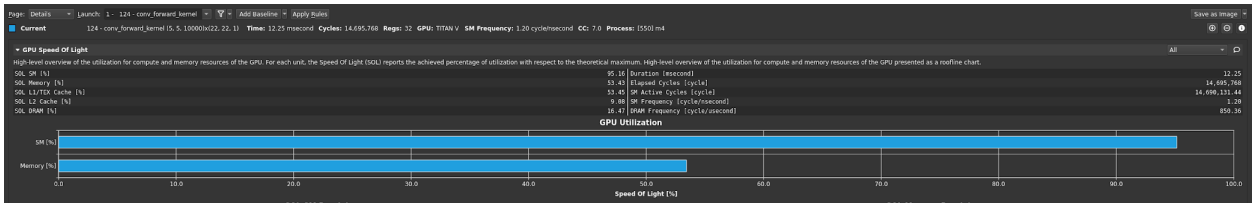


Fig 13. Nsight GPU SOL utilization for optimization 4 with a TILE_WIDTH of 16.

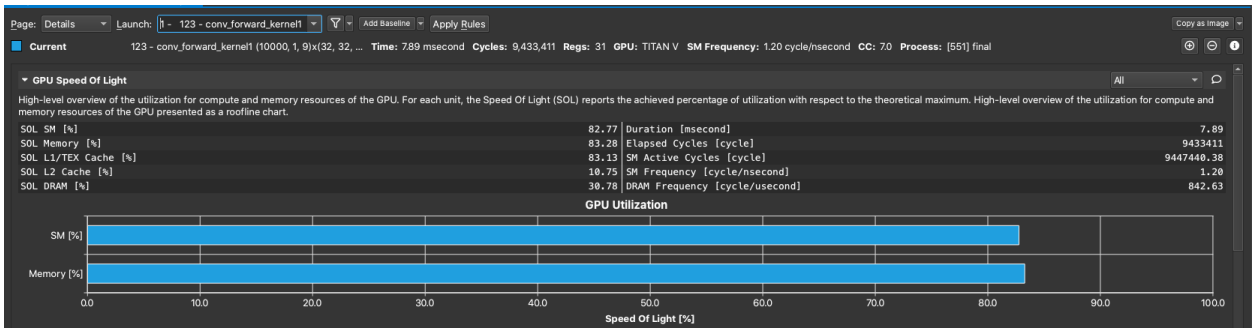
Comparing figures 12 and 13, we can see that optimization 4 performs better as we are using the GPU more efficiently. By loading the input in the shared memory, we have almost doubled the compute performance of the GPU. This leads to the lower Op times as presented in Table 10. Additionally, by examining the dimensions of the inputs involved, we determine a TILE_WIDTH of 10 will perform better than a TILE_WIDTH of 16. Implementing 10 as the TILE_WIDTH drops the Op times by an additional 7.742 ms and results in a best runtime of 73.519 ms.

Optimization 7: Multiple Kernels for Different Convolution Layers

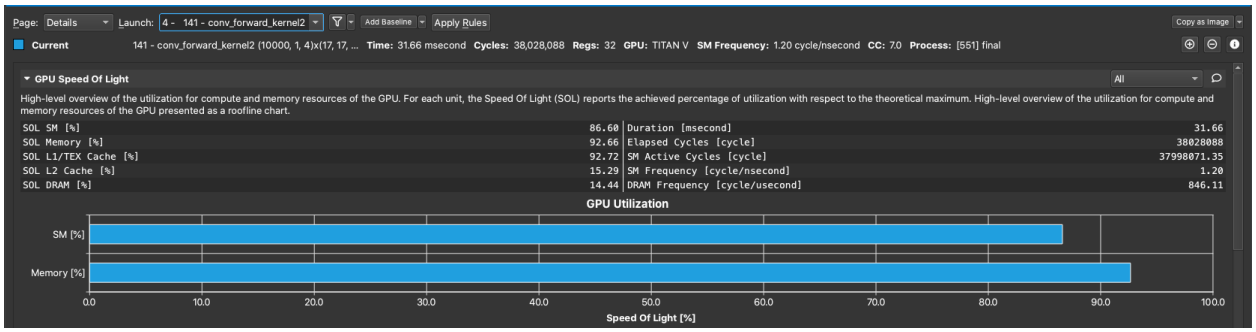
For this optimization method, we actually used our findings from Optimization 2 (Using constant memory + varying tile size) and extended it to develop two different kernels for the two different convolution layers. Specifically, we noticed that each layer performed better or worse as we varied tile_size *independent* of one another. This led to the insight that we should treat each convolution layer as a separate kernel, because they had fundamentally different characteristics. Specifically, the first layer only had one channel with 4 output features compared to the second layer which had 4 channels and 16 output feature maps. We believed that defining a custom kernel for each layer would be very beneficial to the operation times because we no longer needed to conduct ghost operations for a layer which didn't need such large tiles to begin with. Having implemented this, we saw the following operation times as outlined in Table 11 with nsight analysis shown in Figure 14.

Batch Size	**Baseline Operation Times**	**Multiple Kernels for Different Convolution Layers**
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: 0.085696 ms Layer 1 OpTime: 0.119743 ms Layer 1 LayerTime: 6.862659 ms Layer 2 GPUTime: 0.320957 ms Layer 2 OpTime: 0.352253 ms Layer 2 LayerTime: 5.362552 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: 0.812858 ms Layer 1 OpTime: 0.840154 ms Layer 1 LayerTime: 62.389112 ms Layer 2 GPUTime: 3.29252 ms Layer 2 OpTime: 3.316904 ms Layer 2 LayerTime: 51.076183 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: 7.221705 ms Layer 1 OpTime: 7.26804 ms Layer 1 LayerTime: 636.066702 ms Layer 2 GPUTime: 31.557038 ms Layer 2 OpTime: 31.623949 ms Layer 2 LayerTime: 523.825365 ms

Table 11: Operation times comparing GPU convolution with baseline kernel vs. Multiple kernels for different convolution layers



(a)



(b)

Fig 14. Nsight GPU SOL analysis for Layer 1 (a) and Layer 2 (b). Using multiple kernels for each layer enables us to use different TILE_WIDTH values for each layer. In this case, we get significant speedup in Layer 1 by using a TILE_WIDTH of 32 and in Layer 2 by using a TILE_WIDTH of 17.

Therefore, as it can be seen from both the operation times and the nsight analysis, our hypothesis was correct in that by defining a kernel for each layer we were able to see tremendous benefit from this method. The one thing to note from Figure14 is that our memory and GPU utilization was very similar to one another for both layers meaning that we didn't have a scenario where we were wildly mismanaging the memory bandwidth nor compute resources. Thus, for our final implementation of the code to climb up the ranks, we combined this code with Optimizations 4,6,8 and,9 to drive down the operations times.

Optimization 8: Tuning for Restrict + Loop Unrolling

Another optimization method we believed would help in decreasing the operation times would be to take advantage of loop unrolling and using the *restrict* keyword. We thought this optimization would be fruitful because of the underlying compiler optimizations. Specifically, using the *restrict* keyword helps compiler optimization by removing pointer aliasing [7], and loop unrolling helps the compiler parallelize the loops. However, upon implementing the *restrict* keyword, we did not see any performance improvement. We think this is related to the GPU architecture (i.e., Volta architecture) used for the project. Volta architecture no longer has a

read-only cache, instead, it has a general L1 data cache [8], so code without the restrict commands also resides in the L1 cache. Additionally, we got almost no performance improvement when we just added #pragma unroll keywords before our for loops but we saw significant improvement (~30ms) when manually unrolling the loops for the data to load from global memory to shared memory. We think this is because compilers unroll the loops for performance improvement without explicit unroll keywords, but cannot unroll *all* loops, even with unroll keywords. As such, manually unrolling the loop provided the additional performance benefit. Thus, we implemented this method in our final implementation of the code.

Optimization 9: Exploiting Parallelism Optimization

Finally, for our last optimization technique, we took advantage of the parallelism offered by CUDA and defined the thread and block dimensions such that for each image, we have independent convolution computations for both input and output channels. This optimization wasn't very insightful as it took advantage of the parallelism in the GPU architecture that we've been learning throughout the course. However, combining this method with Optimizations 4,6,8 offered very fruitful results that are shown in Table 12.

Batch Size	**Final Implementation Operation Times**
100	Layer 1 GPUTime: 0.073792 ms Layer 1 OpTime: 0.128736 ms Layer 1 LayerTime: 8.472932 ms Layer 2 GPUTime: 0.267838 ms Layer 2 OpTime: 0.292254 ms Layer 2 LayerTime: 5.242714 ms
1000	Layer 1 GPUTime: 0.68374 ms Layer 1 OpTime: 0.721436 ms Layer 1 LayerTime: 63.859618 ms Layer 2 GPUTime: 3.042894 ms Layer 2 OpTime: 3.075693 ms Layer 2 LayerTime: 47.838421 ms
10000	Layer 1 GPUTime: 7.381248 ms Layer 1 OpTime: 7.430304 ms Layer 1 LayerTime: 635.139296 ms Layer 2 GPUTime: 29.312321 ms Layer 2 OpTime: 29.357537 ms Layer 2 LayerTime: 462.58922 ms

Table 12: Final Optimization Operation Times

Team Organization + Work Splitting

Finally, for Milestone 5, we divided the work in the following manner: Prerit was responsible for optimization 4, Jeonghyun was responsible for optimization 5, and Sagar was responsible for optimization 6 but each team member could obviously reach out to others on the team to ask for help/ideas. Once we finished the three required optimizations, we then began to focus on climbing the rankings with further improved optimizations. As such, Jeonghyun also

worked on optimizations 7 and 8. Finally, we all contributed to the report equally and worked intimately with each other to help with various bugs and fixes. Overall, the team worked very well together!

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

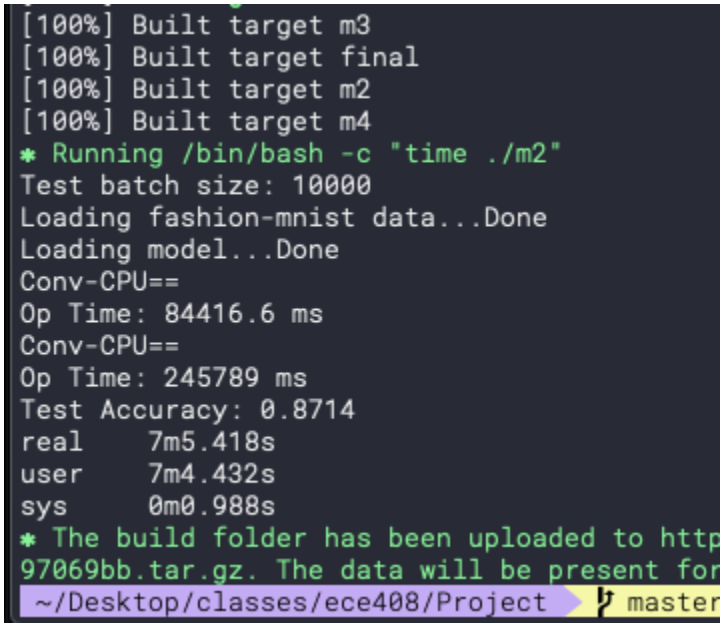


Figure 1: RAI output for Mini-DNN on CPU with CPU Convolution

- 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.

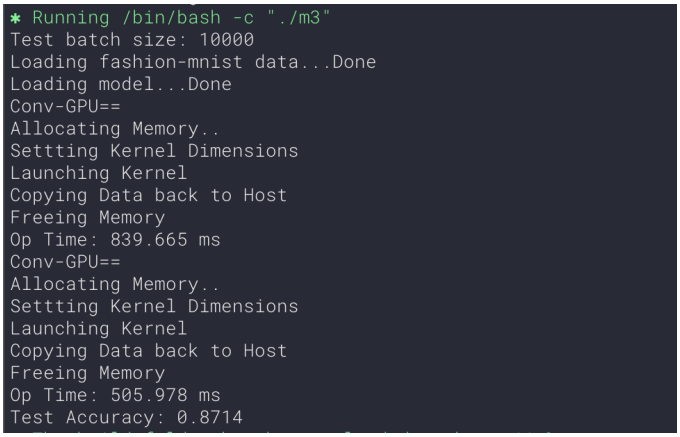


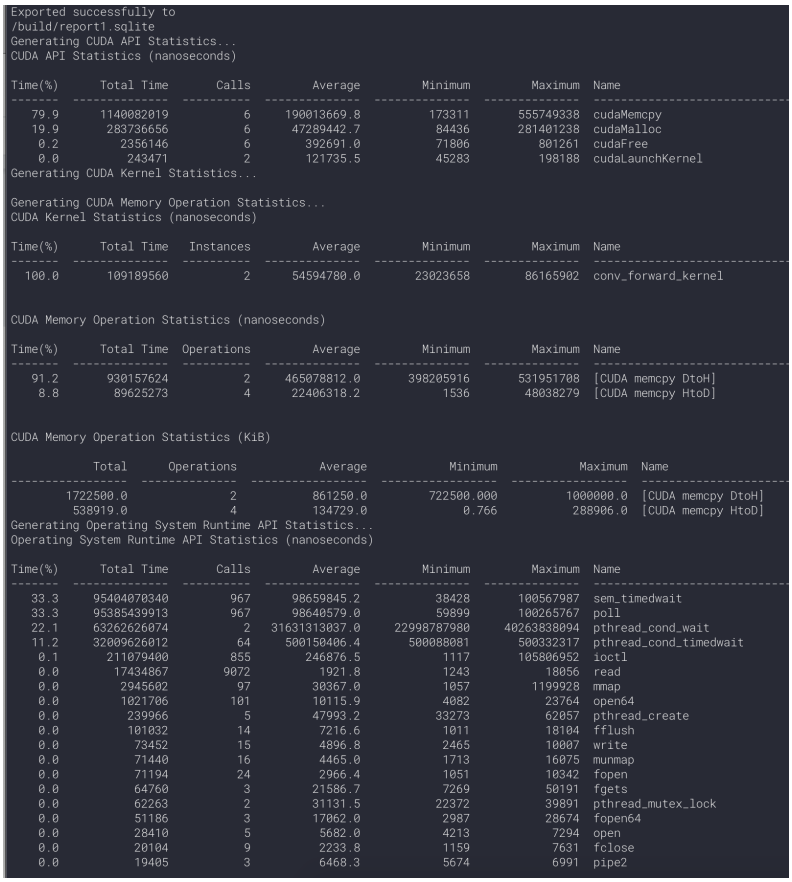
Figure 2: Figure 1: RAI output for Mini-DNN on CPU with GPU Convolution

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:

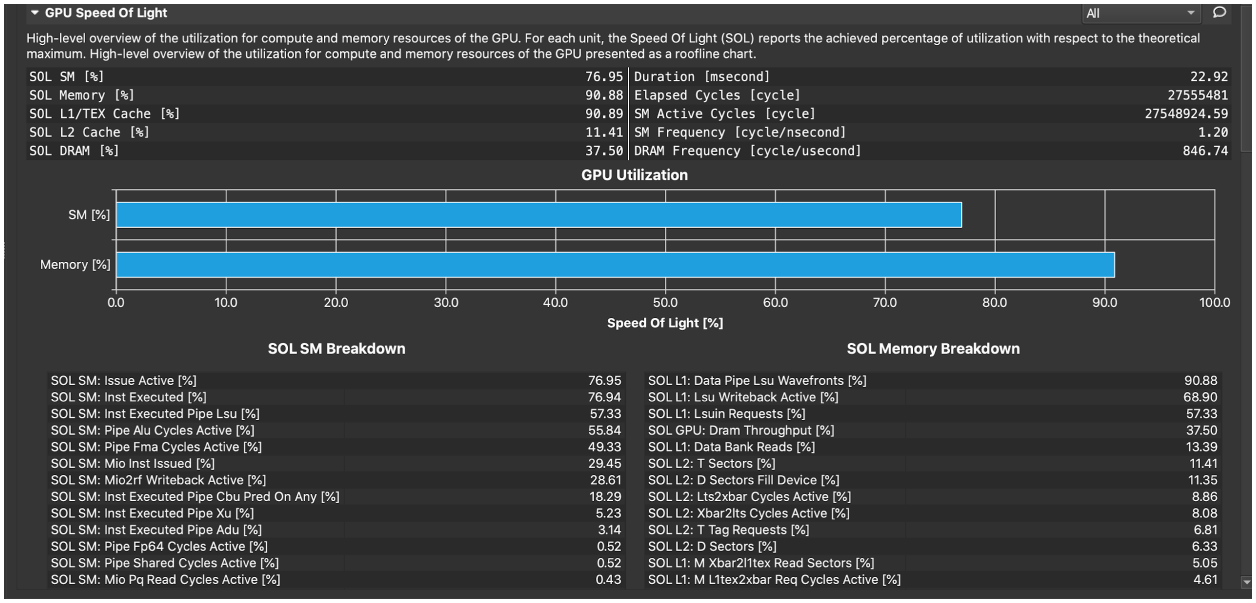


Figure 4: Nsight Stats for GPU Baseline Convolution for batch size = 10000

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 GPUTime: 0.303806 ms Layer 1 OpTime: 0.331198 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 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.

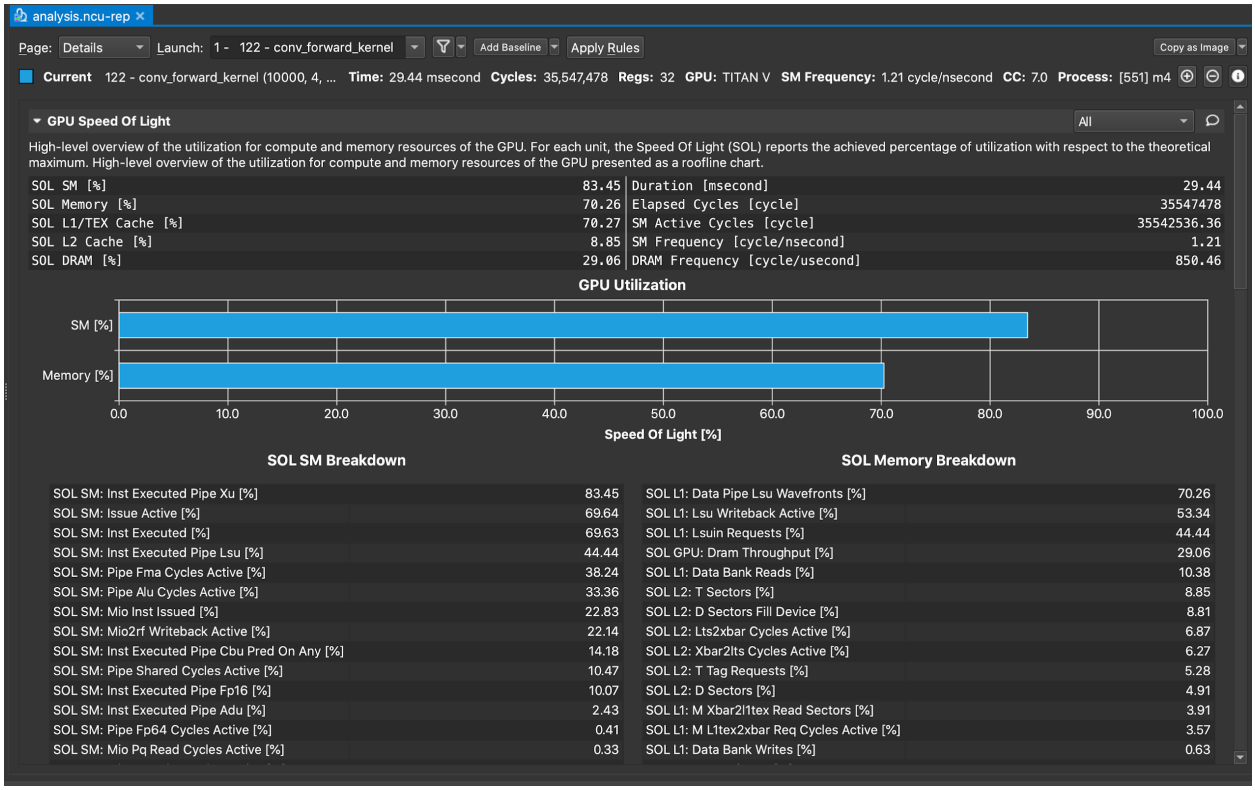


Figure 5: Nsight Stats for GPU Convolution with FP16 Optimization for batch size = 10000

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.

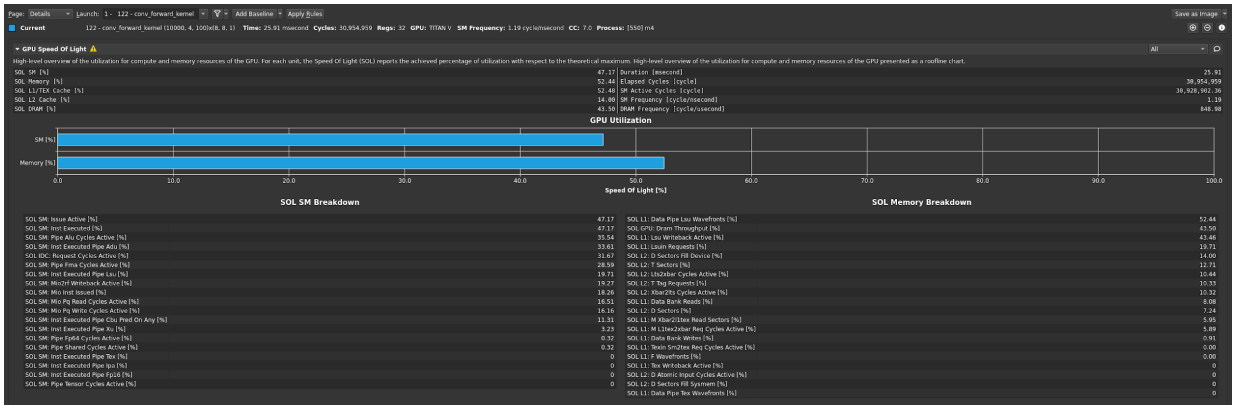


Figure 6: Nsight Stats for constant memory GPU Convolution with TILE_WIDTH = 8 for batch size = 10000

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.

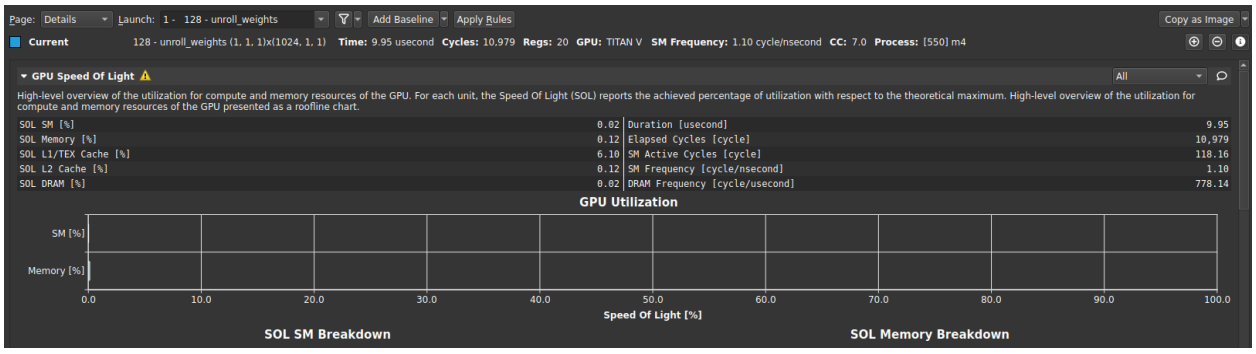


Figure 6: Nsight Stats for unrolling weights kernel

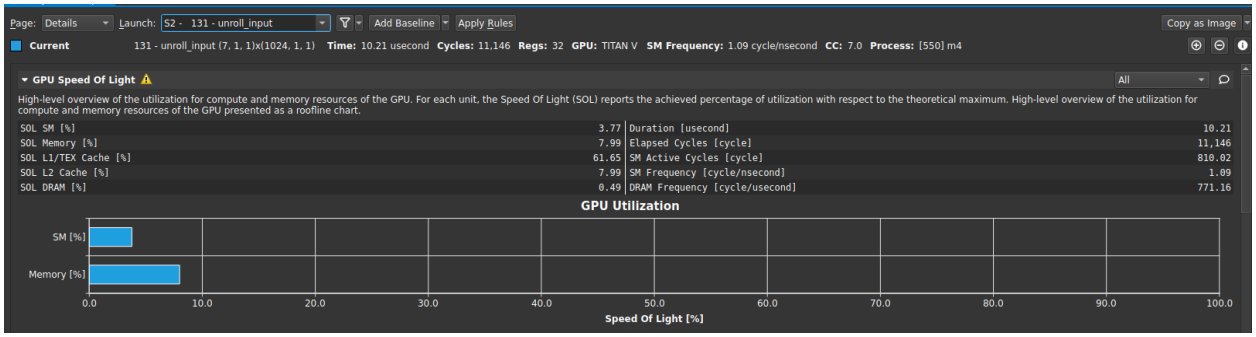


Figure 7: Nsight Stats for unrolling input kernel

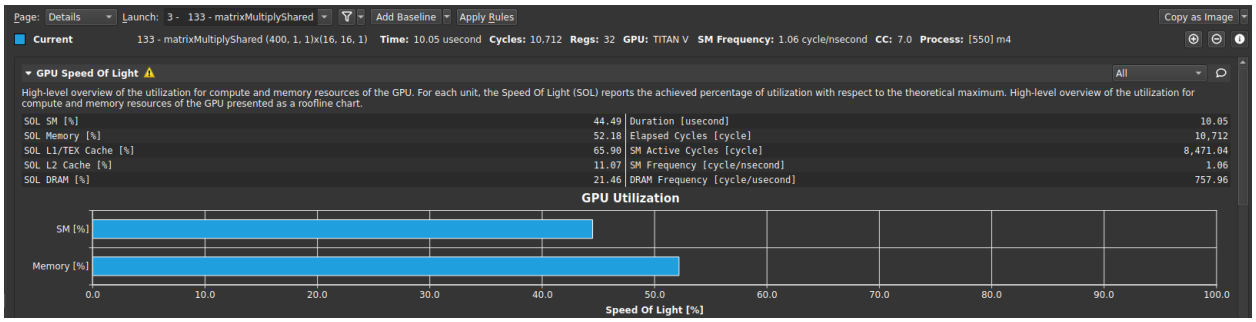


Figure 8: Nsight Stats for shared matrix multiply kernel

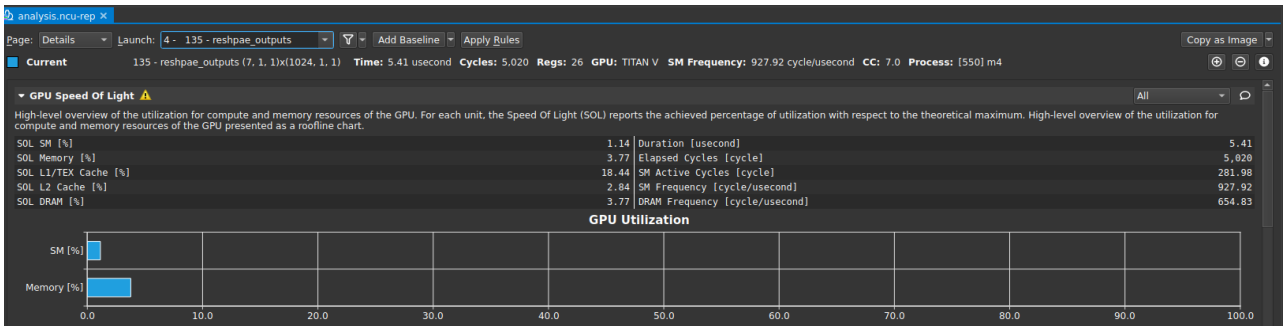


Figure 9: Nsight Stats for reshape kernel

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.
[6] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
[7] https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/
[8] https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf